Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

PhD Thesis

# Cache based Optimization of Stencil Computations
# An Algorithmic Approach

submitted by

Mohammed Shaheen

submitted

November 6, 2013

Supervisor / Advisor

Prof. Dr. Christian Theobalt
Prof. Dr. Hans-Peter Seidel

Reviewers

Prof. Dr. Christian Theobalt

Prof. Dr. Hans-Peter Seidel
Dr. Robert Strzodka

delivered by
EG EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

**Date of Colloquium:**
5 November 2013

**Dean:**
Univ.-Prof. Dr. Mark Groves Faculty of Natural Sciences and Technology I
Department of Computer Science
Saarland University
Saarbrücken, Germany

**Examination Board:**
Prof. Dr. Philipp Slusallek (Chair)
Saarland University

Prof. Dr. Hans-Peter Seidel
Max Planck Institut Informatik

Prof. Dr. Christian Theobalt
Max Planck Institut Informatik

Dr. Robert Strzodka
Nvidia Corporation

Dr. James Tompkin
Max Planck Institut Informatik

## Abstract

We are witnessing a fundamental paradigm shift in computer design. Memory has been and is becoming more hierarchical. Clock frequency is no longer crucial for performance. The on-chip core count is doubling rapidly. The quest for performance is growing. These facts have lead to complex computer systems which bestow high demands on scientific computing problems to achieve high performance.

Stencil computation is a frequent and important kernel that is affected by this complexity. Its importance stems from the wide variety of scientific and engineering applications that use it. The stencil kernel is a nearest-neighbor computation with low arithmetic intensity, thus it usually achieves only a tiny fraction of the peak performance when executed on modern computer systems. Fast on-chip memory modules were introduced as the hardware approach to alleviate the problem.

There are mainly three approaches to address the problem, cache aware, cache oblivious, and automatic loop transformation approaches. In this thesis, comprehensive cache aware and cache oblivious algorithms to optimize stencil computations on structured rectangular 2D and 3D grids are presented. Our algorithms observe the challenges for high performance in the previous approaches, devise solutions for them, and carefully balance the solution building blocks against each other.

The many-core systems put the scalability of memory access at stake which has lead to hierarchical main memory systems. This adds another locality challenge for performance. We tailor our frameworks to meet the new performance challenge on these architectures. Experiments are performed to evaluate the performance of our frameworks on synthetic as well as real world problems.

## Abstract

Wir erleben gerade einen fundamentalen Paradigmenwechsel im Computer Design. Speicher wird immer mehr hierarchisch gegliedert. Die CPU Frequenz ist nicht mehr allein entscheidend für die Rechenleistung. Die Zahl der Kerne auf einem Chip verdoppelt sich in kurzen Zeitabständen. Das Verlangen nach mehr Leistung wächst dabei ungebremst. Dies hat komplexe Computersysteme zur Folge, die mit schwierigen Problemen aus dem Bereich des wissenschaftlichen Rechnens einhergehen um eine hohe Leistung zu erreichen.

Stencil Computation ist ein häufig eingesetzer und wichtiger Kernel, der durch diese Komplexität beeinflusst ist. Seine Bedeutung rührt von dessen zahlreichen wissenschaftlichen und ingenieurstechnischen Anwendungen. Der Stencil Kernel ist eine Nächster-Nachbar-Berechnung von niedriger artihmetischer Intensität. Deswegen erreicht es nur einen Bruchteil der möglichen Höchstleistung, wenn es auf modernen Computersystemen ausgeführt wird.

Es gibt im Wesentlichen drei Möglichkeiten dieses Problem anzugehen, und zwar durch cache-bewusste, cache-unbewusste und automatische Schleifentransformationsansätze. In dieser Doktorarbeit stellen wir vollständige cache-bewusste sowie cache-unbewusste Algorihtmen zur Optimierung von Stencilberechnungen auf einem strukturierten rechteckigen 2D und 3D Gitter. Unsere Algorithmen erfüllen die Erfordernisse für eien hohe Leistung und wiegen diese sorgfältig gegeneinander ab.

Das Problem der Skalierbarkeit von Speicherzugriffen führte zu hierarchischen Speichersystemen. Dies stellt eine weitere Herausforderung an die Leistung dar. Wir passen unser Framework dahingehend an, um mit dieser Herausforderung auf solchen Architekturen fertig zu werden. Wir führen Experimente durch, um die Leistung unseres Algorithmen auf syntethischen wie auch realen Problemen zu evaluieren.

# Summary

Multi- and many-core architectures are rapidly becoming the norm in high performance computing. The trend towards many-core architectures exacerbates the problem because the increasing number of on-chip parallel cores renders an exponential growth in the compute capability whereas the system bandwidth increases only linearly. This shaped the so-called *memory wall* problem. Small, yet fast, on-chip memories called *caches* were introduced as the hardware approach to mitigate this problem.

At the core of the memory wall problem in scientific computing are iterative loops over discrete local operators. A typical representative is a stencil computation with constant weights or a sparse matrix vector product in case of variable weights. This computation pattern achieves only a tiny fraction of the peak computational performance due to its low arithmetic intensity.

In the literature, this problem has been approached from three perspectives. The first is cache-aware whereby the cache parameters are known to the algorithm at either compile or run time. The second is the cache-oblivious approach which, in contrast to the cache-aware approach, does not assume any knowledge about the cache parameters. The third looks at the problem as nested loops and uses loop transformation frameworks to optimize them.

All approaches revolve around the idea of cache locality optimization which exploits the fact that data is moved to the on-chip cache memories before any computation is done on it. Once data is on-chip, these approaches devise different techniques to perform as much computation on it as possible before it gets evicted from the cache. Cache locality can be improved by partitioning, also called *tiling*, the domain on which the stencil operator is applied into small groups called *tiles*. Tiles may also span the time which is the number of iterations in iterative stencil computations. However, as each iteration depends on the previous one, tiles must be skewed in the iteration space to form the so-called *time skewing*. The granularity of execution is the tile which means that the processor does not proceed to the next tile until it has finished executing the current tile. The advantage of running the stencil on tiles rather than the domain as a whole is that if the tiles are so small that they fit into the on-chip cache, the algorithm runs on data stored in the cache whose bandwidth is faster than the main memory bandwidth. Ongoing research is being conducted to design optimal tile parameters (shape and size) that are ideally also executable in parallel to cope with the multi- and many-core revolution.

While different cache optimized approaches have been tried in the past, they have never been so successful. In this thesis, we pinpoint the deficiency of the state-of-the-art approaches and envision a set of cache-aware and cache-oblivious algorithms which avoids the

deficiency. For example, on a quad-core Xeon X5482 3.2GHz system, a synthetic machine peak benchmark reaches 40.8 GFLOPS. On a large 3D domain of $504^3$ double precision floating point and 100 iterations, a hand-vectorized single-threaded naive stencil implementation achieves 1.6 GFLOPS and there is no improvement in the multi-threaded version because the system memory bandwidth limits the performance. A state-of-art automatic loop transformation framework Pluto [7] achieves 1.9 GFLOPS for this stencil computation with four threads. In comparison, our schemes, called cache-aware time skewing (*CATS*) and cache-oblivious parallelograms in iterative stencil computations (*CORALS* ) perform, in average, already at 5.3 GFLOPS with a single thread. Their performance soars to 13.0 GFLOPS with four threads. Furthermore, CORALS scores an excellent result on 2D domains by reaching 47% of the machine peak benchmark.

Efforts to improve the scalability of memory accesses in multiprocessing systems have introduced the non-uniform memory access (NUMA) memory systems whereby memory is physically distributed but logically shared. As such, in these architectures memory access time depends on the memory location relative to the processor, i.e. whether the memory location is in the local memory (memory modules directly connected to the processor) or in the remote memory (memory modules which are connected to other processors). The performance of our schemes, CATS and CORALS, may deteriorate depending on the amount of remote memory data that has to be updated by a certain core. This poses another challenge for performance on the new NUMA systems and schemes targeting these architectures have to explicitly account for a new performance aspect called *data-to-core affinity*. To this end, we have extended CATS and CORALS to meet the new performance challenge. Our new schemes targeted at NUMA architectures which we call *nuCATS* and *nuCORALS* attain striking absolute performance and salient scalability on these architectures.

In summary, this thesis contributes to both cache-aware and cache oblivious-stencil computations. It has been perceived that high performance stencil computations are attainable solely by locality optimizations. While this is partially true as locality optimization yields the lion's share of performance, other factors such as parallelism and data-to-core affinity are essential for performance and their absence may adversely impact the performance gains expected from locality optimizations. This thesis formulates the requirements to achieve high performance stencil computations. It shows how failure to reckon with any of these requirements leads to schemes which hardly surpass the performance of an optimized naive scheme without any locality optimizations. We introduce two novel schemes which cater for all requirements simultaneously and achieve outstanding performance benefits from locality optimizations on stencil computations. We also show how to face the performance challenge posed by the NUMA memory systems in many-core systems and devise two new schemes that exhibit outstanding performance scaling.

## Acknowledgements

This thesis would not come to be without the help and support of the following people. I would like to express my sincere gratitude to my supervisor Prof. Dr. Hans-Peter Seidel who gave me the great opportunity to work and do my research at the inspiring environment of the MPI Informatik and the Computer Graphics Group.

I would like to thank my direct mentor Dr. Robert Strzodka who sparked my interest in High Performance Computing. Robert continuously provided me with incitement and encouragement. On the other hand, I would like to thank my direct supervisor Prof. Dr. Christian Theobalt. Christian tirelessly provided me with all support, help, and feedback to finish this thesis with his group. I am indebted to both of them for their advice and guidance in my research.

I am grateful to Dr. Dawid Pajak for the long hours of fruitful discussions. Finally, I would like to thank my parents for their continuous support, and Ola, my wife, for her patience all the time.

# Contents

**1 Introduction** **1**
1.1 Problem Statement . . . . . 2
1.2 Motivation . . . . . 3
1.3 Contributions . . . . . 3
1.4 Thesis Outline . . . . . 5
1.5 List of publications . . . . . 6

**2 Fundamentals** **9**
2.1 Memory Hierarchy . . . . . 9
2.2 Locality Principle . . . . . 10
2.3 Memory Configurations in Multiprocessing Systems . . . . . 11
2.4 Stencil Computations . . . . . 12
2.5 Space-time Traversals . . . . . 14

**3 Related Work** **19**
3.1 Cache Aware Stencil Optimization . . . . . 19
3.2 Cache Oblivious Stencil Optimization . . . . . 21
3.3 Loop nest Optimization . . . . . 22

**I Iterative Stencil Computations for SMP Systems** **23**

**4 Cache Accurate Time Skewing** **25**
4.1 Previous Work . . . . . 25
4.2 Contributions . . . . . 26
4.3 Cache Accurate Time Skewing (CATS) . . . . . 27
4.4 Results . . . . . 35
4.5 Conclusion . . . . . 42

**5 Performance Modelling** **45**

5.1 Hardware Setup . . . 45
5.2 Software Setup . . . 46
5.3 Naive and Time Skewed Stencil Computations . . . 48
5.4 Varying Cache Size . . . 50
5.5 Performance Model . . . 52
5.6 Model Evaluation . . . 54
5.7 Conclusion . . . 57

**6 Cache Oblivious Parallelograms 59**
6.1 Previous Work . . . 59
6.2 The Cache Oblivious Parallelograms Algorithm . . . 60
6.3 Results . . . 69
6.4 Conclusion . . . 74

**II Iterative Stencil Computations for NUMA systems 77**

**7 NUMA Aware Stencil Computations 79**
7.1 NUMA-aware CATS Scheme (nuCATS) . . . 80
7.2 NUMA-aware CORALS Scheme (nuCORALS) . . . 81
7.3 Results . . . 84
7.4 Conclusions . . . 99

**III Application 101**

**8 Optical Flow Estimation from RGBZ Cameras 103**
8.1 Optical Flow from RGBZ Images . . . 104
8.2 Minimisation . . . 108
8.3 Implementation . . . 109
8.4 Scene Flow Derivation . . . 110
8.5 Evaluation . . . 110
8.6 Conclusion . . . 113

**9 Conclusion and Future Work 115**

**Bibliography 121**

# Chapter 1

# Introduction

Ever since the advent of the first general purpose electronic computer, computer technology has witnessed a continuous evolution. This evolution has been made possible by the advances in hardware and innovations in computer design. In 1945, the mathematician and computer scientist **John von Neumann** proposed a model for computer architecture called the *stored program computer* which assumes that an instruction fetch and a data operation cannot occur simultaneously because they share a common bus. Therefore, he proposed to use a single storage structure to hold both data and instructions. This is referred to as the *Von Neumann bottleneck* and often limits the performance of the system. Later on, **Harvard Mark I** proposed another architecture called *Harvard architecture* in which he suggested to separate storage and signal pathways for both data and instructions. Modern computer systems combine aspects from both architectures; on chip small, yet fast cache memories are provided between the processor and the main memory to circumvent the performance bottleneck of the Von Neumann architecture. These cache memories are often separate and have separate access pathways for data and instructions, the so-called *Modified Harvard architecture*. On the other hand, the Von Neumann architecture is used for the off chip main memory access.

While clock frequencies of the processors increase, memories have not been able to keep pace with this increase. The growing disparity of speed between the processor and the off chip memory bandwidth is often referred to as the *memory wall* problem. In response, dual-, triple- and quad-channel memory interfaces have been introduced. They alleviate the problem temporarily but their scaling is too expensive to keep up with the exponentially growing number of cores. Data intensive applications suffer from severe slow down from this problem. In contrast to *system bandwidth* (off chip bandwidth), the aggregate cache bandwidth scales naturally with the number of cores if each core has a separate connection to its cache. Then doubling the number of cores also doubles the number of

connections and thus the aggregate cache bandwidth. Ideally we would like data intensive applications to scale with the cache bandwidth rather than the system bandwidth. To this end, data intensive applications must utilize the *cache locality principle* as much as they can to break off the dependency on system bandwidth and scale with cache bandwidth. One basic computational pattern that suffers from the slow system bandwidth and can draw great performance benefits from the cache locality principle is *stencil computations*.

## 1.1 Problem Statement

Stencil computations are a class of kernels which update each point in a grid with a linear or non-linear combinations of its neighbor values, for example with a weighted subset of its neighbors. These kernels are usually applied many times (iterations) on the grid, therefore they are called *iterative stencil computations*. The number of neighbouring points involved in the update of a grid point including itself is called the *stencil size*. The distance between the updated grid point and the farthest grid point in the neighbourhood involved in the update is called the *stencil order*. Equation (1.1) shows a model stencil update in 3D. The equation is characterized by low arithmetic intensity (13 floating point operations) as the case for typical stencil computations. However; on a $504^3$ grid and a Xeon X5482 machine, 4 threads and 100 iterations of this stencil problem achieve less than 4% of the computational stencil peak from registers. The reason is the memory wall problem. To analyze how the memory wall could have such an adverse impact on the performance, we first briefly explain how data processing happens in modern architectures.

$$
\begin{aligned}
X^{t+1}_{i,j,k} = c_1 \cdot X^t_{i-1,j,k} + c_2 \cdot X^t_{i,j-1,k} + c_3 \cdot X^t_{i,j,k-1} \\
+ c_4 \cdot X^t_{i+1,j,k} + c_5 \cdot X^t_{i,j+1,k} + c_6 \cdot X^t_{i,j,k+1} \\
+ c_0 \cdot X^t_{i,j,k}
\end{aligned}
\tag{1.1}
$$

Instructions fetched from instruction cache are executed by the processor until an instruction operating on data stored in the main memory is reached. The processor first checks the data cache for the data, and initiates a so-called *cache miss* if the needed data is not available in the cache. It temporarily stalls waiting for the larger and slower memories to provide the data which is then stored in the cache for the probability that it will be needed in the future (locality principle, see Chapter 2). The processor then resumes executing instructions. Given the discrepancy of speed between the processor and memory, the processor has to burn many cycles waiting for the cache to be refilled with the needed data which hurts the performance. The time needed to refill the cache with the needed

data is called *cache miss penalty*. Iterative stencil computations on huge domains that do not fit into the cache undergo the same cache miss penalty for each iteration. For the very first iteration data is stored in the main memory and has to be fetched to the cache once needed. However, if the domain size is too big to fit entirely into the cache, the cache is overwritten already from the same iteration and data from the same address has to be fetched over and over again for each iteration which interprets the paradoxical stencil computations performance. In this thesis, we approach this problem from an algorithmic point of view, we observe the naive stencil performance on the most up-to-data architectures, analyze the problem and propose an algorithmic solution.

## 1.2 Motivation

The motivation for this thesis is the importance of the stencil computation pattern. This importance stems from the importance of the stencil kernel which is a very frequent computational kernel arising in a variety of important scientific and engineering applications. The stencil kernel constitutes a significant fraction of the execution times of these applications. Such applications comprise partial differential equation solvers, image processing, computer vision, and simulations of climate, weather, and ocean. For example in the image processing domain, the well-known variational methods which often boil down to a system of equations that has to be solved numerically are a good example of stencil codes.

Although the problem has been thoroughly investigated and many approaches for alleviating it have been proposed, all proposed approaches are either complicated which limits their applicability or unsatisfactory in terms of the performance gains compared to what the state-of-art hardware can offer.

## 1.3 Contributions

The idea behind all approaches for optimizing stencil computations is to group the domain (space) and the stencil iterations together to shape the *iteration space* or *space-time*. This iteration space is then partitioned into groups on which the stencil is applied. The partitioning and the resulting groups of space-time are referred to as *tiling* and *tiles*, respectively. The application of the stencil on the tiles happens in an atomic fashion, i.e. the whole tile (space and time) is executed by one processor before it can proceed to the next tile. In the literature, different tiling approaches are examined and they differ by tile size, tile shape, the way stencil is executed on each tile, or other parameters which will be explained later in this thesis. In Chapter 2, we explain how tiling is used to mitigate the stencil problem in more detail.

In this thesis, we propose a set of high performance stencil algorithms that perform beyond system bandwidth limitations. We approach the stencil problem from two perspectives. The first assumes that the cache parameters are available at execution time and use them to optimize the tile size so that the iterative stencil application incurs as few as possible cache misses. In the literature, such algorithms are usually referred to as *cache-aware algorithms* as opposed to *cache-oblivious algorithms* whereby no knowledge about cache parameters is assumed. The second perspective is the cache-oblivious approach. We present our cache-aware and cache-oblivious algorithms for stencil computations which are targeted at *symmetric multiprocessing* (SMP) in the first part (**Part I**) of this thesis. Symmetric multiprocessing systems (SMP) are characterized by the uniform main memory access, i.e. memory access time does not depend on the location of memory relative to the processor as opposed to the *non-uniform memory access* (NUMA) systems whereby memory access time depends on the location of memory relative to the processor. In the second part (**Part II**), an extension of the previous cache-aware and cache-oblivious algorithms to NUMA systems is presented. An important computer vision application which uses stencil computation at the core is presented in the third part (**Part III**). In particular, we present a novel high performance variational framework for scene flow estimation from one depth and one color cameras. In summary, our major contributions are:

- A cache-aware framework for iterative stencil computations called CATS [58, 60].
- A cache-oblivious framework for iterative stencil computations called CORALS [59].
- A performance model which predicts the execution time of the CATS scheme based on the number of incurred cache misses [57].
- Extensions of both CATS and CORALS (called nuCATS and nuCORALS, respectively) which exhibit weak as well as strong scalability when the memory is logically shared but physically distributed, the so called **NUMA** architectures [52].
- A novel variational scene flow estimation from color and depth cameras that can benefit from the above algorithms.

The presented stencil algorithms can be applied to stencils of any order and size. Moreover, the stencil coefficients may vary across the domain, i.e., the schemes support also a product with a sparse banded matrix.

### 1.3.1 Limitations

The presented algorithms are targeted at iterative stencil computations on structured rectangular grids. This means that high performance can be drawn from our algorithms

when the stencil is applied multiple times on the domain. Also, our algorithms can not be applied to stencil computations on non-structured grids or structured grids on arbitrary domain shapes (e.g. circular domains), although they can be easily extended to suit the latter type of domains.

## 1.4 Thesis Outline

The rest of this thesis is structured as follows. Chapter 2 introduces the memory hierarchy which is regarded as the hardware approach to mitigate the memory wall problem. In this Chapter, we also distinguish two milestones in the memory architecture which influence the performance scalability of the memory bound algorithms in general. The first is symmetric multiprocessing (*SMP*) architectures and the second is the non-uniform memory access (*NUMA*) architectures. We also present the stencil problem and the previous efforts to approach the problem. In this context, we present *cache blocking*, *time skewing*, and *cache oblivious* stencil computation as the fundamental approaches to tackle the stencil problem. The rest of this thesis is organized into the following parts.

**Part I** is particularly devoted to our cache-aware and cache-oblivious stencil algorithms for symmetric multiprocessing systems (SMP). In Chapter 4, we propose a cache-aware framework for stencil computations called *cache accurate time skewing (CATS)*. The CATS scheme mitigates all drawbacks of the state-of-art approaches to optimize stencil computations on multidimensional domains and delivers high speedups over a recent code transformation tool and an optimized naive approach.

In Chapter 5, we present a performance model for the naive and the CATS schemes which uses the number of incurred cache misses to estimate the expected execution time on a certain machine. We use the model to examine the impact of scaling system and cache bandwidths on the naive and the CATS schemes for iterative stencil computations. We purport that scaling the cache bandwidth is more important to squeeze performance from time skewed iterative stencil computation algorithms.

Chapter 6 introduces our cache-oblivious parallelograms for iterative stencil computations (*CORALS*) algorithm. We show how CORALS caters for data locality, parallelism, and vectorization simultaneously in a cache-oblivious fashion. This high performance scheme comes at the cost of an irregular work-load distribution for which we introduce a tightly integrated load balancer to ensure a high utilization of all resources. At the time of publishing the related paper, CORALS delivered unprecedented outstanding performance which was not achievable with previous cache-oblivious stencil algorithms.

**Part II** is devoted to the variants of our cache-aware and cache-oblivious stencil algo-

rithms for non-uniform memory access (NUMA). In Chapter 7, we present *nuCATS* and *nuCORALS* as the NUMA variants of the CATS and CORALS schemes. We examine the scalability behavior of both CATS and CORALS on architectures with non uniform memory access (NUMA) memory systems, elicit the requirements for scalable high performance schemes on the NUMA architectures, and design variants for both CATS and CORALS that meet all these requirements. The presented comprehensive performance and scalability comparisons against various benchmarks derived from machine characteristics and a state-of-art code transformer and stencil compiler show the superiority of nuCATS and nuCORALS and emphasize the importance of considering the NUMA aspect as a very important one in any stencil code designated for scalable high performance computing.

**Part III** is devoted to applications. In particular we propose a novel variational framework for computing the scene flow from an RGB image sequence and the geometric information obtained from an active range sensor (Chapter 8).

We conclude this thesis in Chapter 9 and propose future directions for the research on this topic.

## 1.5 List of publications

The work presented in this thesis has been published in the following papers:

- **NUMA Aware Iterative Stencil Computations on Many-Core Systems**
  *Mohammed Shaheen, Robert Strzodka*
  The 26th IEEE International Parallel & Distributed Processing Symposium (2012)
- **Cache Accurate Time Skewing in Iterative Stencil Computations**
  *Robert Strzodka, Mohammed Shaheen, Dawid Pajak, Hans-Peter Seidel*
  The 40th International Conference on Parallel Processing (2011)
- **Impact of System and Cache Bandwidth on Stencil Computation Across Multiple Processor Generations**
  *Robert Strzodka, Mohammed Shaheen, Dawid Pajak*
  The 2nd Workshop on Applications for Multi and Many Core Processors (2011)
- **Time Skewing Made Simple**
  *Robert Strzodka, Mohammed Shaheen, Dawid Pajak*
  The 16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 2011)
- **Cache Oblivious Parallelograms in Iterative Stencil Computations**

*Robert Strzodka, Mohammed Shaheen, Dawid Pajak, Hans-Peter Seidel*
The 24th ACM/SIGARCH International Conference on Supercomputing (2010)

- **Overcoming Bandwidth Limitations in Visual Computing**
  *Robert Strzodka, Mohammed Shaheen, Dawid Pajak*
  Visual Computing Research Conference (2009)

# Chapter 2

# Fundamentals

From vacuum tubes to transistors, and from integrated circuits to very large scale integration (VLSI) and ultra large scale integration (ULSI), computers have witnessed notable advances since their inception. In this chapter, we elaborate on some concepts and features of modern computer systems which are very important to understand this thesis. In particular we explain the hierarchical nature of computer memories nowadays. We also define the locality principle which is crucial for performance and discuss the two modern alternatives for organizing multiprocessing systems. The last Section is dedicated for stencil computations and the fundamental approaches to optimize their performance.

## 2.1 Memory Hierarchy

The memory system in nowadays machines is a hierarchy of memory modules with different costs, capacities, and speeds. The closer a memory module to the processor, the faster, smaller, and the more expensive it is. On the top of the hierarchy, reside the fastest memory modules which are called the *CPU registers*, see Figure 2.1. Registers typically hold the most frequently used data, e.g. loop variables. On the next level reside bigger memory modules called the *cache memories* which are located on-chip and hold a subset of the data and instructions stored in the relatively slow main memory. The cache itself is a hierarchy with typically two or three levels. Each cache level typically holds a subset of the data stored in the next level and is smaller in size. The main memory (RAM) is located off chip and serves as a staging area for data stored on the slow hard disks.

For programmers, understanding how data is moved up and down the hierarchy is crucial for writing high performance codes since well-written computer programs tend to access data stored at a certain level of the hierarchy more often than they access the next lower level. The idea here is to exploit the fact that data must be moved to the top of the

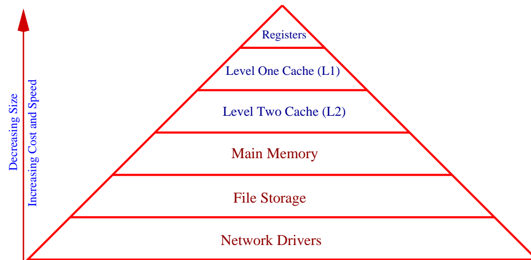



**Figure 2.1.** The Memory Hierarchy. Memory speed and cost increase while size decreases in the arrow direction. On chip memory is written in blue, and off chip memory is written in red.

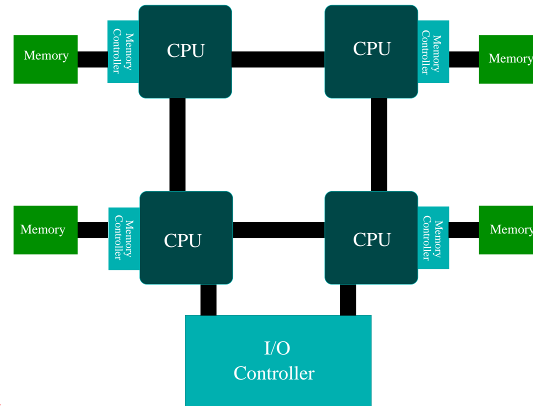



**Figure 2.2.** Non Uniform Memory Architecture (NUMA). Each processor has its own memory but can also access memories owned by other processors. All processors share the same address space.

hierarchy, typically registers and caches, before the first operation is performed on it. To this end, the program has to be structured in such a way that allows it to perform all required operations on the subset of data available in the cache. This relieves the processor from stalling each time it needs any data item from that subset and does not find it in the cache. This is called the *cache locality* principle and is discussed in detail in Section 2.2. Overall, the memory hierarchy serves as a large block of memory which provides data to the processor at the speed of the fast memory near the top of the hierarchy and at the cost of the cheap memory near the bottom of the hierarchy.

## 2.2 Locality Principle

The cache memory is designed based on two concepts; the *spatial* and *temporal* locality of data access. *Spatial locality* principle implies that if some data are referenced at any point in time, it is likely that nearby data are also used in the computation. *Temporal locality* implies that data referenced now are likely to be used again in the future.

The idea here is that when a processor needs data at a certain memory address, it first checks the on chip memory, typically L1 cache, if data is not found there, a so-called L1 *cache miss* occurs and data is sought in L2 cache, and again if it is not found, L2 cache miss occurs, and data is sought in the higher cache levels (typically L3) if available before it is sought in the main memory and then higher levels of the hierarchy. Data are then moved up on the hierarchy. Typically a block whose size equals cache line size and contains the sought data is fetched to the cache. On one hand, fetching data in blocks satisfies the spatial locality principle. On the other hand, keeping data in the cache

satisfies the temporal locality principle. When data is sought and found in any cache level, a so called *cache hit* happens and there is no need to search further down in the hierarchy which adversely impacts the performance of the application. Our goal in this thesis is to improve the performance of stencil computations by maximizing the cache hit rate and minimizing the cache miss rate by maximizing data reuse once it is on chip (in cache).

Cache misses can be classified into

- Compulsory misses. The first access to data is always not in the cache, a compulsory miss occurs and a block containing the sought data must be brought into the cache.
- Capacity misses. The cache size is limited so it cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being evicted and fetched again later. The goal of the work presented in this thesis is to minimize this class of cache misses.
- Conflict misses. When two or more blocks of data frequently accessed by a program map to the same cache line, a so called conflict miss occurs and entails the so called *cache thrashing*. A lot of research is conducted to minimize this class of cache misses on both hardware and software levels; however, this is outside the scope of this thesis.

## 2.3 Memory Configurations in Multiprocessing Systems

The memory in a cluster of microprocessors can be configured in one multiprocessing systems in two ways; non-uniform memory access (NUMA) and symmetric multiprocessing (SMP). In NUMA architectures [48], memory is physically distributed but logically shared meaning that each processor has its own local memory but also can access memory owned by other processors. Symmetric multiprocessing (SMP) systems, on the other hand, use a common bus to access a single shared main memory in the multiprocessing machine architectures. However, when new processors are added to the machine, this bus can get overloaded and thus becomes a performance bottleneck. Large multiprocessor systems use NUMA to alleviate the latter problem in which processors can access their local memories quickly compared to accessing memories owned by other processors which comes with interprocessor communication overhead. While this can significantly improve performance as long as data are localized, on the downside applications which need data movement between processors can suffer from severe performance slowdowns.

Maintaining coherent caches in NUMA systems has a notable overhead. The reason is that cache coherent non uniform memory access systems (ccNUMA) resort to interprocessor

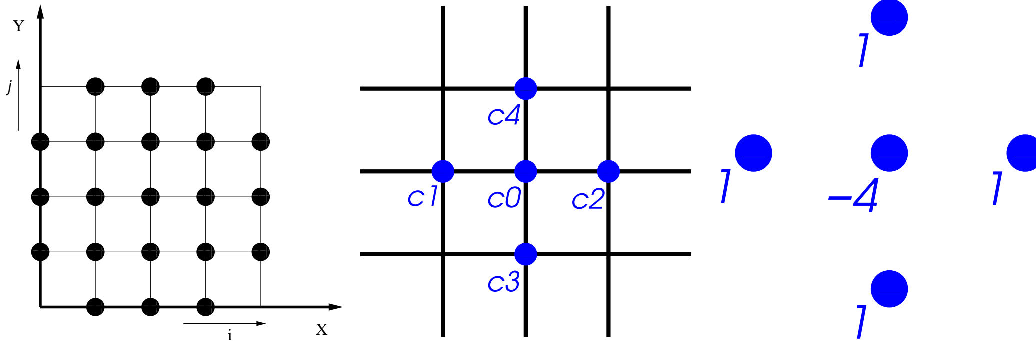



**Figure 2.3.** The 2D grid can be used to discretize Poisson Equation.

**Figure 2.4.** A 2D 5-point stencil with constant weights.

**Figure 2.5.** The 2D stencil resulting from discretizing Poisson equation.

communication between cache controllers to maintain a consistent image of the cached memory locations among the caches.

Each processor with its local memory is called a NUMA node. Data is moved on the bus between the clusters of NUMA nodes using scalable coherent interface (SCI) technology which maintains a coherent cache or consistent data in the caches of the different processors [62]. Although the NUMA nature of an architecture is transparent, programmers can effectively exploit the so called *first-touch allocation* [5] of operating systems to effectively optimize their programs for this type of architectures. First touch allocation means that a memory location is allocated in the local memory of the processor on which the process which has touched that memory location first is running. In other words, when a process running on processor $x$ accesses for the first time (no other process has accessed it before) a variable, e.g. for initialization, a *page fault* [62] happens since the memory has not yet been allocated for the variable. The operating system allocated the memory on the local memory of processor $x$. We show how we benefit from this feature to improve the scalability of stencil computations in Chapter 7.

## 2.4 Stencil Computations

As mentioned in the Introduction (Chapter 1), stencil computations are a class of iterative kernels which update each point in a grid with a weighted subset of its neighbours usually multiple times, and they are referred to as *iterative stencil computations*.

Stencil computations are ubiquitous in scientific computing primarily because the action of discretized local differential or integral operators can be expressed in this form. For example, using finite differences [41] to discretize the Poisson equation $\Delta u = f$ on a 2D

grid yields for each node $(i, j)$ on the grid (Figure 2.3)

$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = h^2 f_{i,j} \tag{2.1}$$

This can be represented as a linear system of equations in the form

$$Au = b \tag{2.2}$$

Solving large PDEs and large linear equation systems in reasonable time requires iterative solvers, e.g. Gauss Seidel solvers [2]. Thus stencil computations are performed repeatedly for many iterations until convergence.

**Algorithm 1** Iterative stencil computations in 2D. The grid shown in Figure 2.4 has the dimensions XSIZE and YSIZE. The outer loop (t-loop) denotes the number of iterations needed for convergence in case of e.g. a numerical solver. c0-c4 are the stencil weights

```
Iterative_2D_Stencil ()
{
  for(t = 0; t < T; t++) {
    for(j = 0; j < YSIZE; j++) {
    for(i = 0; i < ZSIZE; i++) {
      A(i,j) = c0*A(i,j) + c1*A(i-1,j) + c2*A(i,j-1) + c3*A(i,j-1) + c4*A(i,j+1)
    }}//x,y
  }//t
}
```

Most often stencil computations are just a linear weighting of a small domain neighborhood, See Figures 2.4 and 2.5. In this case, the stencil computation represents a matrix vector products, with the position dependent stencil weights forming the rows of the matrix, and the discrete values of the domain (the grid) forming the vector. The arithmetic intensity [26] of such an operation is very low, with just one multiplication and addition per every vector and stencil component read, cf. Algorithm 1. Even if the stencil weights are constant and can be stored locally, there are still just two operations per every memory read, whereas even latest triple channel architectures prefer values of 8 and above, e.g. to balance computation and bandwidth on an Intel Core i7: 4(cores)·3.2(GHz)·2(SSE mad double2)/25.6GB/s = 8mad/8B there should be 8 multiply and add (mad) operations for every double (8B). So only a non-linear stencil computations with many operations could prevent it from being memory bound.

When the discrete vector is so small that it fits into the processor's caches, the cache bandwidth becomes the limiting factor (unless there are not enough parallel compute units). However, this is not always the case since stencils in scientific computing typically operate on data much bigger than the cache capacity.

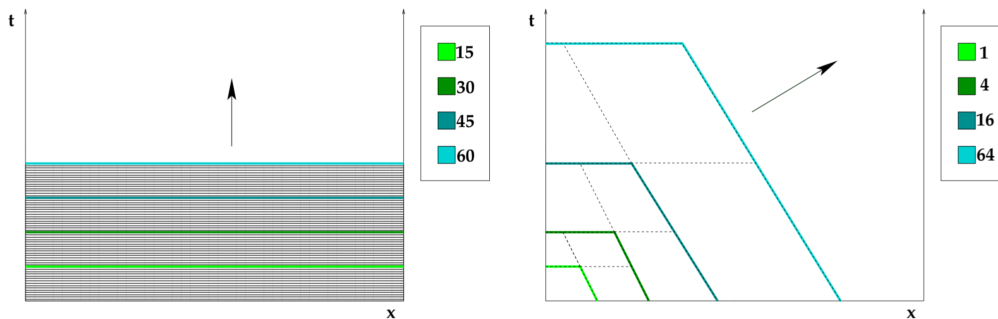



**Figure 2.6.** Naive space-time traversal. The entire domain progresses one timestep after another. The evolution of the surfaces separating the computed and not computed part of the space-time is shown.

**Figure 2.7.** Hierarchical, cache oblivious space-time traversal. It computes on small tiles but the execution order groups them into bigger and bigger structures further improving the volume to surface ratio.

## 2.5 Space-time Traversals

This section describes the different space-time traversal algorithms. Given a discrete $d$-dimensional spatial domain $\Omega := \{1, \ldots, W_1\} \times \ldots \times \{1, \ldots, W_d\}$ with $N := \#\Omega$ values we want to apply the stencil $S : \{-s, \ldots, +s\}^d \times \Omega \to \mathbb{R}$ repeatedly to the entire domain $T$-times. So our space-time domain is given by $\Omega \times \{0, \ldots, T\}$ with the initial values at $\Omega \times \{0\}$ and boundary values at $\partial\Omega \times \{0, \ldots, T\}$, $\partial\Omega := \{0, W_1 + 1\} \times \ldots \times \{0, W_d + 1\}$. Let us also define $N_s$ as the number of non-zero elements in the stencil.

In the space-time $\Omega \times \{0, \ldots, T\}$, there are $TN$ values to be computed, and each output values requires $N_s$ input values, so in the worst case we will have $TNN_s$ accesses to main memory. In this we assume a perfect cache of size $Z$ with one word per cache line, full associativity and optimal replacement strategy, so a cache miss (and thus an access to main memory) occurs if and only if the required value is not present in the cache at that time.

If we access values from timestep $t-1$ to compute values at timestep $t$ then, irrespective of the scheme, we need to store two copies of $\Omega$ during the stencil application. Some stencil computations like Gauss-Seidel [2], that use values from timestep $t-1$ and $t$ while computing timestep $t$, can be performed in-place with just one copy of $\Omega$, but we assume the general case here. A consequence of capturing the entire computation state in two copies of $\Omega$ is, that if these two copies fit into the cache, then just $N$ cache misses will be encountered no matter how large $T$ is.

### 2.5.1 Naive Stencil Computations

Given values at $\Omega \times \{t-1\}$ for any $t \in \{1, \ldots, T\}$, we can compute values at the next timestep $\Omega \times \{t\}$ with $d$ nested spatial loops with index $x_k, k \in \{1, \ldots, d\}$ running from 1 to $W_k$ respectively. In practice, the inner most loop $x_1$ corresponds to the unit stride direction, and the stride increases with every additional outer loop. Embedding all spatial loops in an outer time loop with index $t$ running from 1 to $T$, we traverse the space-time in a naive way, progressing with the entire domain one timestep after another. Figure 2.6 visualizes this procedure.

In order to expand the computed part of the space-time without additional reads from main memory we would need to cache the entire surface between the computed and not computed part, i.e. the entire $\Omega \times \{t\}$ would have to reside in cache, which is only possible for fairly small $\Omega$. However, partial data reuse during the spatial traversal of $\Omega \times \{t\}$ occurs naturally whenever lower dimensional sub-structures of $\Omega \times \{t-1\}$ of size $W_1, W_1 \cdot W_2, \ldots, W_1 \cdots W_{d-1}$ stay in the cache after their first use, e.g. in 2D for a 3x3 stencil the line $\{1, \ldots, W_1\} \times \{5\} \times \{t-1\}$ is needed for the computation of the three lines $\{1, \ldots, W_1\} \times \{4, 5, 6\} \times \{t\}$. Nevertheless, even with big caches this scheme performs poorly because the data reuse occurs only between data of adjacent timesteps.

### 2.5.2 Cache Blocking

Cache blocking [21] is a cache utilization algorithm that attempts to minimize the cache misses of a program and hence enhance the performance by increasing the use of data already present in the cache. Cache blocking can yield a significant performance boost on computation patterns which are characterized by the large discrepancy between the floating point operations count and memory references needed. The basic idea here is to apply the computation on small blocks which are contiguous in memory. A loop nest which appears e.g. in matrix-matrix multiplication requires referencing memory locations that are not on the unit-stride dimension causing frequent cache misses when the array sizes exceed the cache size. Cache blocking restructures the loop nest so that the computation proceeds on small blocks which fit in the cache. Blocking variants and optimal blocking parameters have been a hot topic for research for many years.

Iterative stencil computations draw some benefit from cache blocking. Stencil computations are in essence a sparse matrix vector product. Cache blocking restructures computation to proceed on blocks of contiguous memory locations. However, in iterative stencil computations, only one iteration can be applied to each block since data at the boundary of each block depends on each other which limits the reuse from cache to the spatial locality. A technique to use the temporal locality and thus increase the overall reuse is

called *Time Skewing*. In time skewing, multiple iterations can be applied to each block by skewing the boundary of each block in time which is explained in the Section 2.5.3.

### 2.5.3 Time Skewing

When advancing certain blocks of the domain several stencil iterations ahead of the rest, we need to respect data dependencies induced by the form of the stencil. So called *time skewing* techniques have been described by Wolf [71], Song et al, [54] and Wonnacott [72]. Thereby, the time axis corresponds to the number of iterations the stencil is applied to the entire spatial domain, in general this is not the same as the time parameter of the PDE or other computational process in which the stencil is applied.

With this additional time axis we can form the space-time domain $\Omega \times \{0, \dots, T\}$, where the data at $\Omega \times \{0\}$ is given and the task is to compute a value for all remaining points in the space-time, see Figure 2.6. The idea is to look at the entire iteration space formed by the space and time, the *space-time*, and divide it into tiles that can be executed quickly in cache. To proceed to the next step locally without access to main memory, stencil computations require the neighbors of the previous iteration, therefore the tiles in the space-time are skewed with respect to the time axis, see Fig. 2.7. The execution inside the tile is very fast because these values are produced and consumed on-chip without the need for a main memory access. Only at the tile boundaries additional data must be brought into cache. If we know the L2 cache size in our CPU we can choose the tile size such that the base of the tile fits into the cache. To optimize for the memory hierarchy, we could further subdivide each tile into small tiles, whose bases fit into the L1 cache. All such parameters must be set conservatively, because any overestimation leads to cache thrashing and a severe performance penalty. In a multi-threaded environment, it is difficult to find the right parameters, since the available cache can be shared by different application threads and it is not clear which portion of the cache is available to the stencil computation at any given time. The tile dimensions form a large optimization space which can be explored empirically [32] and systematically [35, 49], whereby it makes a big difference if the exploration targets mainly data locality, or parallelism, or both equally.

### 2.5.4 Cache Oblivious Stencil Computation

A fourth approach is to use a hierarchical tiling that adapts automatically to the available cache size, which is thus named as *cache-oblivious* [17]. This can be achieved by iteratively tiling the space-time until small tiles are reached, see Figure 2.7. This hierarchical tiling has the advantage of optimizing for the whole memory hierarchy even without explicitly knowing its parameters (number of levels, size of each level,..etc). Despite the massive

cache miss reduction induced by cache-oblivious schemes, practically translating that into performance is fairly difficult. The programmer has to find a good tradeoff between the overhead associated with deepening the recursion tree (cache oblivious schemes are often implemented as a recursive algorithm) and the performance gains from running the scheme on data stored on deeper levels (closer to the processing elements) of the memory hierarchy.

# Chapter 3

# Related Work

Naive implementations of stencil computations suffer heavily from system bandwidth limitations. Most approaches to optimize stencil computations look at the problem from a cache locality perspective and therefore propose optimization techniques that can be classified into two categories. The first assumes that the cache parameters (cache size, cache hierarchy, and cache line size) are known beforehand, hence algorithms from this category are often referred to as *cache aware algorithms*. The second category is called *cache oblivious algorithms*. Although cache oblivious algorithms do not assume any knowledge about the cache parameters, in the asymptote, they use the cache as competently as the cache aware algorithms [19]. A more general approach, however, looks at the stencil as a special case of a loop nest with data dependencies and proposes loop nest optimizations accordingly. In this Chapter we review the efforts to optimize stencil computations in the literature and underline our work within the different approaches.

## 3.1 Cache Aware Stencil Optimization

Cache blocking techniques have been developed to optimize for the spatial data locality. Lam et al. [36] study the impact of blocking on the performance of a matrix multiplication algorithm and conclude that the performance of blocked algorithms is highly dependent on the block and problem sizes. Kamil et al. [31] present recent empirical results from applying different blocking techniques in stencil computation algorithms. Frumkin and van der Wijngaart [20] have tight lower and upper bounds on the number of data loads. Dursun et al. [16] propose a multilevel parallelization framework for high order stencils based on domain decomposition and massage passing to exchange the subdomain boundaries on a cluster of CPUs. In [13], Datta el al. perform comprehensive stencil computations optimization and auto-tuning with both cache-aware and cache-oblivious approaches on a

variety of state-of-the-art architectures, including GPUs.

Although recent results show large benefits in applying these techniques on multi-core architectures; however, no matter how efficiently we load the data into the caches, for data exceeding the cache size, we still read every vector component at least once per timestep from the main memory and for repeated applications of the stencil, this is far too much. To further reduce access to main memory, we need to exploit the outer loops that repeat the stencil computations over the same domain and make use of temporal locality. The reason is that spatial locality optimizations performed in the aforementioned works remain by construction bounded by the peak system bandwidth. In view of the exponentially growing discrepancy between peak system bandwidth and peak computational performance, this is a severe limitation for all current multi-core devices and even more so for future many-core devices. When advancing certain parts of the domain several stencil iterations ahead of the rest, we need to respect data dependencies induced by the form of the stencil. So called time skewing techniques have been described by Wolf [71], Song et al, [54], McCalpin and Wonnacott [40], and Wonnacott [72]. Thereby, the time axis corresponds to the number of iterations that the stencil is applied to the entire spatial domain, e.g., this can be the explicit time steps of a PDE solver, or the iterations of an iterative solver for linear equation systems. The general idea of time skewing is to tile the space-time into space-time tiles that can be executed with very few cache misses and ideally also in parallel. These requirements lead to skewed tiles in the space- time, see Chapter 2.

Common to all of the above approaches in case of a multi-dimensional domain, is a multi-dimensional tiling strategy: the time and multiple (not necessarily all) spatial dimensions are divided in order to form space-time tiles of approximately the same diameter in all divided dimensions. This minimizes the surface area to volume ratio of the space-time tiles and thus reduces cache misses. It is the best general strategy to traverse a space-time of unknown size [30]. However, knowing the typical cache size per core and domain sizes. In [58] and [60], we contribute an algorithm that does the exact opposite: we tile only one spatial dimension (resulting in enormous space-time tiles) and use the relatively large caches of nowadays cores to reduce the 2D or 3D problem to a 1D problem, where spatial tiling is not necessary and instead a wavefront traversal can be used. The resulting large space-time tiles can not only be traversed in a cache efficient fashion. They also allow for a SIMD friendly computation along the non-tiled unit stride dimension. On the other hand, while previous approaches resort to multilevel tiling to utilize the memory hierarchy, we only tile for the last cache level and thus maintain large tiles and wavefronts. Despite its simplicity, our scheme achieves superior results in comparison to the conventional multidimensional and multilevel tiling schemes, see Chapter 4.

Stencil optimization efforts were not only confined to CPUs, e.g. Williams et al. [68] and Datta et al. [14] investigate various stencil optimization techniques on the Cell BE.

Christen et al. [12] apply various optimization techniques to stencil computations arising in the biomedical simulations. The latter work targets the Cell BE and GPU architectures.

Kamil et al. [33] devise a microbenchmark called *Stanza Triad* to evaluate the impact of modern memory subsystem design on 3D stencil computations. In particular they evaluate the effectiveness of the prefetching engine in cache-based systems. They also develop a proxy for general stencil computations from which they derive a memory cost model for quantifying the performance of cache blocking stencil computations. In Chapter 5, we devise a performance model for naive and time skewing stencil codes based on the number of incurred cache misses. Our performance model provides useful insights into the most effective improvements for stencil computations on future processors.

## 3.2 Cache Oblivious Stencil Optimization

Schemes that are able to exploit the memory hierarchy without explicitly knowing its size are first described in [47] and are called *cache oblivious algorithms*. Frigo and Strumpen [17] introduced a cache oblivious stencil scheme that divides the iteration space recursively into smaller and smaller space-time tiles and thus generates high temporal locality on all cache levels without knowing its sizes. The cache misses are greatly reduced leading to the desired reduction of system bandwidth requirements, however, the performance gains are relatively small in comparison to this reduction. Strumpen and Frigo [56] report a 2.2x speedup against the naive implementation of a 1D Lax-Wendroff kernel on a IBM Power5 system for periodic and constant boundary conditions after optimizing the software aspects of the scheme. After multifold optimizations and parameter tuning Kamil et al. [32] achieve a 4.17x speedup on the Power5 (15 GB/s theoretical peak bandwidth), 1.62x on an Itanium2 (6.4 GB/s) and 1.59x on an Opteron (5.2 GB/s) system for a 7-point stencil (two distinct coefficient values) on a $256^3$ domain for periodic boundary conditions. However, for constant boundary conditions the optimized cache oblivious scheme is only faster on the Opteron achieving a 2x speedup at best. The compared naive code is optimized with ghost cells and compiled with optimization flags.

The above optimizations of the cache oblivious scheme are all directed at single-threaded execution. Frigo and Strumpen later analyzed multi-threaded cache oblivious algorithms [18]. One example deals with the cache misses of a 1D stencil code with parallel tile cuts. Blelloch et al. [6] discuss the construction of nested parallel algorithms with low cache complexity in the cache oblivious model for various algorithms including sparse matrix vector multiplication. However, these are mainly theoretical papers. In [59], we present a new cache oblivious scheme for iterative stencil computations that delivers the high speedups promised by the great cache miss reduction and clearly outperforms more general trans-

formation tools and optimized naive code. The most impressive results of our scheme are achieved in 2D. While a synthetic benchmark iterates over registers and performs 25.1 GFLOPS, our scheme iterates over a gigabyte large domain and performs 19.1 GFLOPS which are about 75% of the benchmark performance. This is an exceptionally outstanding performance as if the system bandwidth is hardly an issue.

## 3.3 Loop nest Optimization

A more general approach to improve the temporal locality of iterative stencil computations is to see them as a special case of perfectly or imperfectly nested loops with data dependencies, see [53, 1, 24]. In this category, optimized and parallelized stencil kernels are automatically generated based on hardware models. Li and Song [38] present a framework for automatic stencil loops tiling based on memory cost analysis from which the tiling parameters (tile size and shape) that minimize the capacity misses are derived. Rivera and Tseng [51] develop a cost model for selecting tiling parameters and use it to find an optimal tiling transformation for 3D Jacobi kernels. Christen et al. [11] present an autotuner and a code generation framework for parallel stencil code based on domain specific description of the stencil kernel. The user of the framework can either choose from a set of predefined strategies or design a custom one to find optimal optimization on the machine in use. The polyhedral model provides an abstraction for valid transformations of nested loops. For an automatic source-to-source translation three steps are required: dependence analysis, transformations in the polyhedral model and code generation. Bondhugula et al. [7] present a complete system called PluTo [45] comprising all three steps. Given a source file it generates the optimized transformed code that can be compiled instead of the original source. Obviously, PluTo cannot successfully exploit data dependencies hidden behind complex index or pointer arithmetic, but it performs very well when arrays are allocated statically and data dependencies are expressed clearly. Other state-of-art tiling schemes for nested loops are HiTLoG [27, 34] and PrimeTile [46, 25]. Harton et al. [25] compare the performance of these tools. Recently, the Pochoir stencil compiler [63] uses the parallel cache oblivious algorithm [18] to optimize a stencil kernel specified in a domain specific stencil language. Although Pochoir succeeds in leveraging performance from the cache oblivious stencil algorithm of Frigo and Strumpen, the achieved performance is still poor and for certain problem size lies in the vicinity of a carefully optimized naive scheme.

# Part I

# Iterative Stencil Computations for Symmetric Multiprocessing Systems (SMP)

# Chapter 4

# Cache Accurate Time Skewing

In this Chapter, we present our cache aware time skewing scheme (CATS) for symmetric multiprocessing (SMP) memory systems. CATS breaks the memory wall for a certain class of iterative stencil computations. A stencil computation, even with constant weights, is a completely memory-bound algorithm. For example, for a large 3D domain of $500^3$ doubles and 100 iterations on a quad-core Xeon X5482 3.2GHz system, a hand-vectorized and parallelized naive 7-point stencil implementation achieves only 1.4 GFLOPS because the system memory bandwidth limits the performance. Although many efforts have been undertaken to improve the performance of such nested loops, for large data sets they still lag far behind the performance of a synthetic machine peak benchmark. The state-of-art automatic locality optimizer PluTo [7] achieves 3.7 GFLOPS for the above stencil which constitutes less than 10% of the measured computational peak benchmark on the same machine (40.8 GFLOPS). CATS, on the other hand, achieves 13 GFLOPS which is 32% of the peak.

## 4.1 Previous Work

For small discrete vectors that fit into the processor's caches, the cache bandwidth is the decisive factor of performance, but stencils in scientific computing typically operate on data much bigger than the cache capacity. Substantial work has been performed to optimize the data locality in such cases up to the point where tight lower and upper bounds on the number of data loads can be given [20]. Recent results show large benefits in applying these techniques on multi-core architectures [31]. But no matter how efficiently we load the data into the caches, for data exceeding the cache size, we still read every vector component at least once per timestep from the main memory and for repeated applications of the stencil, this is far too much. To further reduce access to main memory, we need to exploit

the outer loops that repeat the stencil computations over the same domain and make use of *temporal locality*. When advancing certain parts of the domain several stencil iterations ahead of the rest, we need to respect data dependencies induced by the form of the stencil. So called *time skewing* techniques have been described by Wolf [71], Song et al, [54] and Wonnacott [72]. Thereby, the time axis corresponds to the number of iterations that the stencil is applied to the entire spatial domain, e.g., this can be the explicit time steps of a PDE solver, or the iterations of an iterative solver for linear equation systems.

With this additional time axis we can form the space-time domain $\Omega \times \{0, \ldots, T\}$, where the data at $\Omega \times \{0\}$ is given and the task is to compute a value for all remaining points in the space-time, see Figure 4.1. Now, the general idea of time skewing is to tile the space-time into space-time tiles that can be executed with very few cache misses and ideally also in parallel. These requirements lead to skewed tiles in the space-time, see Figure 4.2. The tile dimensions form a large optimization space which can be explored empirically [32, 14, 70] and systematically [35, 49, 43], whereby it makes a big difference if the exploration targets mainly data locality, or parallelism, or both equally. A more general approach for optimizing iterative stencil computations is to use a loop transformation and parallelization framework [22, 34, 7, 25, 4]. We compare our results against one of them in detail, namely PluTo [7], which is an easy-to-use fully automatic tool and a good indicator of the performance that can be achieved immediately on these nested loops without any further user interaction.

## 4.2 Contributions

In case of a multi-dimensional domain, all schemes mentioned in Section 4.1 resort to the so called *multi-dimensional tiling* strategy: the time and multiple (not necessarily all) spatial dimensions are divided in order to form space-time tiles of approximately the same diameter in all divided dimensions, our scheme [60], in contrast, tiles only one spatial dimension (resulting in enormous space-time tiles) and use the relatively large caches of nowadays cores to reduce the 2D or 3D problem to a 1D problem, where spatial tiling is not necessary and instead a wavefront traversal can be used. *Multi-dimensional tiling* minimizes the surface area to volume ratio of the space-time tiles and thus reduces cache misses. It is the best general strategy to traverse a space-time of unknown size [30]. However, knowing the typical cache size of 128KiB–4MiB per core and domain sizes (100–1000)$^d$, $d = 2, 3$, our scheme creates very large tiles which are much larger than the cache size and not only processes them in a cache efficient, but also a SIMD friendly manner.

Another difference is the treatment of the memory hierarchy. Previous approaches use a *multi-level tiling* strategy: they hierarchically subdivide the space-time tiles either explic-

itly or automatically with the idea that the basis of the sub-tiles will fit into a deeper cache level (e.g. L1) and thus the sub-tile will be processed faster. However, considering the concrete bandwidth and compute ratios, we explore the opposite direction of ignoring the memory hierarchy and instead maximizing the wavefront size in the last cache level (L2 in our case). The idea here is that large wavefronts maximize the number of space-time points that are processed on-chip in a *highly regular* fashion, while processing data from the L2 cache is not a big limitation.

We use a vectorized kernel for the actual computation as otherwise, the data processing could not keep up with the bandwidth of the L2 cache and the memory-bound stencil would become unnecessarily compute-bound. In other words, the vectorization ensures that the kernel remains memory-bound but cannot accelerate the execution beyond that.

We keep the rest of the scheme as simple as possible. We use a single form for all tiles and choose a minimalist parallelization approach: the threads are started once at the beginning and are persistent throughout the computation; furthermore the thread to tile assignment is known at compile-time leading to simple synchronization. This simplicity is of high importance since when benchmark performance is sought in applications, code simplicity is of great benefit to the compiler and hardware. Moreover, dynamic load-balancing is not necessary for tiles of equal size, and replacing barrier synchronization by tile-to-tile synchronization minimizes the idle time. As a result, our scheme achieves about 40% of the computational peak of a Xeon machine on 2D domains, whereas the state-of-art PluTo [7] achieves only 20% of this peak on the same 2D domain. This ratio falls off to 32% on a 3D domain of size $500^3$; however, it is still clearly superior to the performance of PluTo which exhibits a sharper fall off achieving only less than 10% of the peak.

## 4.3 Cache Accurate Time Skewing (CATS)

This section describes our new cache accurate time skewing schemes in comparison to the naive scheme. We first describe some specific variants of CATS and then explain how they combine to give the general CATS scheme.

On a discrete $d$-dimensional spatial domain $\Omega := \{1, \dots, W_1\} \times \dots \times \{1, \dots, W_d\}$ with $N := \#\Omega$ values we want to apply a stencil $S : \Omega \times \{-s, \dots, +s\}^d \to \mathbb{R}$ of *order* $s$ repeatedly to the entire domain $T$-times. In case of a constant stencil, $S$ does not depend on $\Omega$ and has a certain number of non-zero values $N_S := \#S$, otherwise we assume that the stencil is position dependent, $S(x) : \{-s, \dots, +s\}^d \to \mathbb{R}, x \in \Omega$ and has the same number of non-zero values $N_S$ for every position, and $N \cdot N_S$ values overall.

Our space-time domain is given by $\Omega \times \{0, \dots, T\}$ with the initial values at $\Omega \times \{0\}$ and

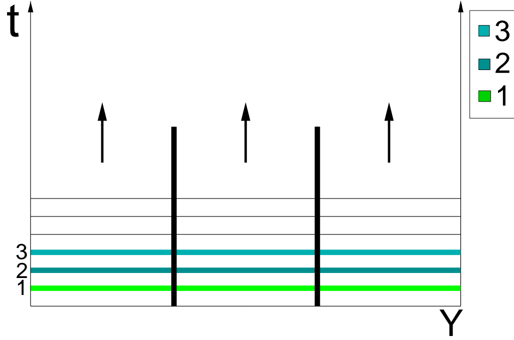



**Figure 4.1.** Naive space-time traversal in parallel with three threads, cf. Alg. 2. Regions of the same color are operated on in parallel, synchronization takes place before starting a different color region. The entire domain progresses one timestep after another in sync in the direction of the arrows. X-dimension goes into the page.

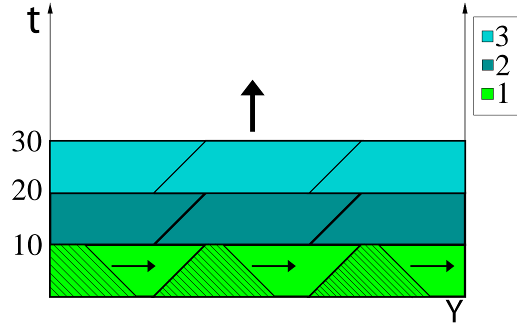



**Figure 4.2.** Cache accurate time skewing with one skewing dimension (CATS1) in parallel with three threads, cf. Alg. 3. Regions of the same color are operated on in parallel, synchronization takes place before starting a different color region. The fine lines show the consecutive wavefront positions and the arrows the traversal direction in each parallelogram. X-dimension goes into the page.

boundary values at $\partial\Omega \times \{0, \dots, T\}$, $\partial\Omega := \{0, W_1+1\} \times \dots \times \{0, W_d+1\}$. In the space-time $\Omega \times \{0, \dots, T\}$ there are $TN$ values to be computed, and each output value requires $N_S$ input values. So in case of a constant stencil we perform $TNN_S$ reads and $TN$ writes; in case of a variable stencil (banded matrix) we perform $2TNN_S$ reads and $TN$ writes.

If we access values from timestep $t-1$ to compute values at timestep $t$ then, irrespective of the scheme, we need to store two copies of $\Omega$ during the stencil application. Some stencil computations like Gauss-Seidel, that use values from timestep $t-1$ and $t$ while computing timestep $t$, can be performed in-place with just one copy of $\Omega$. If these one/two copies of $\Omega$ fit into the cache, then all reads and writes will happen in the cache no matter how large $T$ is. The naive scheme performs much better in this case, as can be seen for the 0.5 million elements case in the Figures. 4.6 and 4.8.

### 4.3.1 No Skewing - NaiveSSE Scheme

The naive stencil implementation has no data reuse between different iterations. The entire spatial domain advances one timestep after another, see Figure 4.1 and Alg. 2. The outermost spatial loop is parallelized with multiple threads, whereby each thread operates on one tile of the domain. The tiles are of the same size so the threads can be synchronized with little overhead after each timestep. The innermost spatial loop (unit stride dimension) is hand-vectorized with SSE2 intrinsics.

**Algorithm 2** The naive scheme for iterative stencil computations in 2D. The spatial domain is cut along the y-dimension into tiles for parallel execution and ystart(tid), yend(tid) are the tile bounds in dependence on the thread ID tid.

```
naive_2D ()
{
  for(t = 0; t < T; t++) {
    for(y = ystart(tid); y < yend(tid); y++) { // parallelized
    for(x = 0; x < WIDTH; x++) {             // vectorized
      apply 2D stencil at position (x,y,t);
    }}//x,y
    synchronize threads;
  }//t
}
```

### 4.3.2 Skewing One Dimension - CATS1 Scheme

**Algorithm 3** CATS1 for iterative stencil computations in 2D. The loop bounds ystart(tid), yend(tid) represent the extent of the tile (parallelogram) along the traversal dimension y. The loop bounds tstart(ts,y), tend(ts,y) represent the extent of the wavefront along the dimension t within the tile, see Figure4.2.

```
CATS1_2D ()
{
  compute height T_Z from cache size (Eq. 4.1);
  for(ts = 0; ts < T/T_Z; ts++) {
    for(y = ystart(tid); y < yend(tid); y++) { // parallelized
      if(y == ystart(tid+1)) {
        wait for (tid+1) to finish its left tile border;
      }
      for(t = tstart(ts,y); t < tend(ts,y); t++) {
        for(x = 0; x < WIDTH; x++) {       // vectorized
          apply 2D stencil at position (x,y-t,t);
        }//x
      }//t
    }//y
    synchronize threads;
  }//ts
}
```

The general idea behind time skewing schemes is to compute multiple timesteps at once in certain parts of the domain thus exploiting the temporal producer-consumer locality. For this purpose we tile one spatial dimension. The plane formed by the chosen spatial dimension and the time dimension is divided into space-time tiles, see Figure 4.2. The tiles are skewed to respect the temporal data dependencies induced by the stencil. Processing within the space-time tile has high temporal locality, while data at the tile borders, in general, has to be reloaded from main memory. Skewed tile borders require more data transfer than straight tile borders. The main decision is on the form of the tiles, aiming

for maximal temporal locality and parallel processing of tiles. We use parallelogram tiles with split-tiling and wavefront processing (Figure 4.2).

These ideas have been described for multiple processors instead of cores already at the onset of time skewing methods by Wonnacott [72], but even in CATS1 we use them differently for multi-dimensional domains. In particular, we show that multi-dimensional tiling of multi-dimensional domains is not necessary. Instead of diagonal wavefronts, we consider axis-aligned wavefronts, and our tile placement is also different. The pipelined temporal blocking by Wittmann et al. [70] and Wellein et al. [67] can also be seen as a variant of space-time wavefront processing. However, they use the term 'wavefront' completely differently, describing the parallelization along the time axis, which benefits from shared caches between multiple threads. This type of 'wavefront' does not exist in our scheme, because we use a different parallelization approach that does not rely on shared caches; instead we construct large space-time wavefronts (using Wonnacott's space-time notion of a wavefront) for the purpose of the data locality maximization.

In wavefront processing we sweep with a skewed space-time surface (the *wavefront*) through the tile along a designated *traversal dimension* (see the arrows in Figure 4.2), maintaining a certain number of the most recent wavefronts in the cache. The computation takes place at the wavefront reusing the data from the previous wavefronts. New data must only be fetched from main memory at the tile borders. For a stencil width of $2s+1$ in the traversal dimension, $2s$ wavefronts plus some temporary variables must reside in an ideal cache for perfect data reuse, but because of limited cache associativity and cache line granularity, a certain value $C_S \in (2s, 2s+1]$ is used in practice, e.g., Wonnacott [72] uses the pessimistic $C_S := 3$ for a 3-wide stencil, we conservatively choose $C_S := 2s + 0.8$ after a cache miss analysis.

The main advantage of wavefront processing is that the tiles can be much bigger than the cache, because only $C_S$ wavefronts must reside in the cache for a perfect producer-consumer locality within the tile. One driving idea behind our cache accurate time skewing schemes is to radically maximize the wavefront size at the expense of any other locality optimizations. In case of one dimensional skewing, CATS1 maximizes the wavefront size such that $C_S$ wavefronts barely fit into the private L2 cache of one thread. Let $Z$ be the size of the private L2 cache and $W_{\max}$ the size of the largest domain dimension, the one to be traversed, then the size of our wavefront is $T_Z N / W_{\max}$ and we can compute the maximal temporal extent of our tile $T_Z$ in dependence on $Z$ as

$$T_Z \quad := \quad \lfloor Z W_{\max} / (C_S N) \rfloor . \tag{4.1}$$

Wonnacott [72] considers diagonal wavefronts $\{(x, y, t) \in \text{Tile} \mid x + y + t = \text{const}\}$ in 2D and concludes that their maximum size in dependence on the domain size makes it

impractical for large domains, so both dimensions must be tiled. The validity of this argument depends on what *large* means. For typical cache and domain sizes, we argue in the opposite direction that a wavefront traversal actually makes multi-dimensional tiling unnecessary. The maximum size of our axis-aligned $\{(x, y, t) \in \text{Tile} \mid y + t = \text{const}\}$ wavefronts grows with the domain size in the same fashion, the growth is proportional to $N/W_{\max}$, but in 2D this is not a big problem even for a small cache of 128KiB, e.g., $3 \cdot 10 \cdot 500 \cdot 8\text{B} = 120\text{KB} < 128\text{KiB}$, which means that on a $500^2$ domain of doubles we could perform $T_Z = 10$ consecutive timesteps in cache. The next section explains that one-dimensional tiling is sufficient even in case of larger (e.g. $10000^2$) domains in 2D and 3D. The reasons for choosing axis-aligned over diagonal wavefronts are the much simpler indexing and more favorable memory access pattern. Axis-aligned refers to the spatial alignment, all wavefronts are always skewed with respect to time.

The time dimension is tiled according to $T_Z$ and Alg. 3 shows the entire CATS1 algorithm in 2D. Figure 4.2 ($T_Z = 10$) shows with thin lines the different positions of the wavefronts and how they progress through the space-time tiles in the direction of the arrows. In CATS1, the parallelization takes place along the same dimension (y-loop in Alg. 4.2) as the wavefront traversal. All threads can start computing concurrently within their parallelograms, there is only a data dependency at the right border of each parallelogram, and thread tid has to wait for thread tid+1 if it reaches its right border faster than tid+1 finished its computation there. For almost all domains the width of the tile is much bigger than its height, so in practice the thread tid does not have to wait. This type of dependence resolution between parallelogram tiles is called *split-tiling* [35]. After completing the wavefront traversal for all tiles in $[0, T_Z)$ the threads are synchronized with little overhead as the tiles are of equal size, and all tiles in $[T_Z, 2T_Z)$ are processed in the same fashion, cf. ts-loop in Alg. 4.2.

Wonnacott [72] and Krishnamoorthy et al. [35] deal with multi-processor systems, so in order to reduce the communication, they align the base of the higher parallelogram with the top of the lower one in the split-tiling scheme. However, this causes load-balancing problems which we avoid by placing the parallelograms simply axis-aligned on top of each other, see Figure 4.2. Because the CPU cores have access to the same main memory, this has no negative effect for us. Irrespective of the parallelogram placement strategy, there is basically no data reuse at the tile borders, because the entire cache is constantly overwritten by the traversing wavefronts.

In 2D and higher dimensions, the innermost loop in CATS1 runs across the entire unit stride dimension (x-loop in CATS1_2D() in Alg. 4.2) so its vectorized execution ensures that the algorithm remains memory-bound when processing data from the L2 cache. In 3D, there are two loops with fixed bounds that span the entire domain. However, these inner loops also mean that more data resides in the wavefront, e.g., the wavefront in 3D

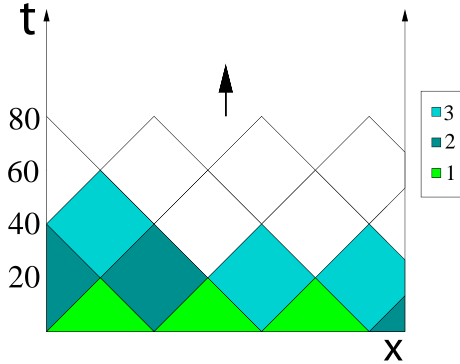



**Figure 4.3.** Cache accurate time skewing with two skewing dimensions (CATS2) in parallel with three threads, cf. Alg. 4. The colors show the a-priori thread to tile assignment, but there is no global synchronization, each diamond waits on the two below. This figure shows the (x,t)-plane for CATS2_2D(), each of the diamonds extends also in the y-dimension (which goes into the page) forming a diamond tube, see Figure 4.4.

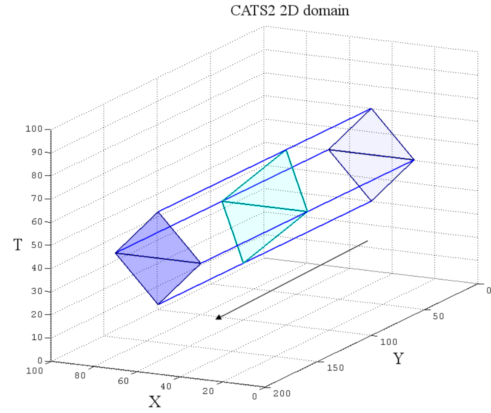



**Figure 4.4.** In CATS2_2D() (Alg. 4) each thread sweeps a diamond-shaped wavefront through a diamond tube region of the space-time. First all values within the current wavefront are computed then the wavefront moves by 1 along the y-dimension. No unnecessary cache misses occur inside the diamond tube although it is much bigger than the cache.

extends in three dimensions (x,y,t)∈[0,WIDTH)×[0,HEIGHT)×[0,$T_Z$). So if WIDTH and HEIGHT are large, the computed $T_Z$ will be smaller than one and we fall back to the naive scheme. Apparently, multi-dimensional tiling of the domain is required in 3D after all, but we present a different solution in the next section.

### 4.3.3 Skewing Two Dimensions - CATS2 Scheme

CATS1 is a special case because it uses the same spatial dimension for tiling and the wavefront traversal. CATS2 and all higher schemes have a distinct traversal dimension and tiling dimensions. For CATS2 one dimension is tiled, and a second is traversed with the wavefronts. This way we reduce the wavefront size in comparison to CATS1 without the need for multi-dimensional tiling.

CATS2 requires two distinct dimensions so it can be applied only in 2D and higher dimensional spatial domains. Figure 4.3 shows the (x,t)-plane with the tiling dimension x in case of CATS2_2D() in Alg. 4. In the (x,t)-plane, the space-time tiles have the shape of diamonds. Together with the traversal dimension (y in 2D), the diamond forms the corresponding space-time tile, a *diamond tube* as depicted in Figure 4.4. The diamonds in Figure 4.3 are the projections of the diamond tubes onto the (x,t)-plane. The processing

**Algorithm 4** CATS2 for iterative stencil computations in 2D. The loop bounds 0, HEIGHT represent the extent of the tile (diamond tube) along the traversal dimension y. The loop bounds tstart(dia,y), tend(dia,y) and xstart(dia,y,t), xend(dia,y,t) represent the extent of the wavefront along the t and x dimension within the tile (diamond tube), see Figure.4.4.

```
CATS2_2D ()
{
  compute diamond size from cache size (Eq. 4.2);
  forall( diamond dia∈diamondSet(tid) ){ // parallelized
    wait on the two diamonds below to finish;
    for(y = 0; y < HEIGHT; y++) {
      for(t = tstart(dia,y); t < tend(dia,y); t++) {
        for(x = xstart(dia,y,t); x < xend(dia,y,t); x++) {
          apply 2D stencil at position (x,y-t,t);//↑vectorized
        }//x
      }//t
    }//y
  }//dia
}
```

of a diamond tube is similar to the traversal in CATS1: a wavefront sweeps through it along the traversal dimension.

Figure 4.4 visualizes the processing of a 2D spatial domain. Therein the diamond tube is a 3D space-time tile, and the wavefront a skewed 2D diamond. For a 3D spatial domain, the diamond tube is 4D and the wavefront is 3D, therefore, the problem is still reduced to a 1D traversal. The key insight is that a wavefront traversal can be performed with a wavefront of arbitrary dimensionality and arbitrary shape. Thus multi-dimensional tiling is not necessary for generating temporal locality and we can process much larger space-time tiles than usual in a cache efficient manner. This is a new idea in wavefront processing of multi-dimensional domains.

We use diamonds in the tiling dimension because of their favorable surface area to volume ratio (cache miss reduction), they are independent of each other when arranged side-by-side (parallel execution), and require only one tile form to cover the plane (simplicity). Orozco and Gao [44] give a quantitative analysis for the first property, however, they use the diamond shape only in 1D with a traditional bottom-up processing of the tile in cache. The second property avoids the problem of dependent tiles encountered by Liu and Li [39], where they have to relax the numerical properties of the scheme in order to gain better parallelization.

As in CATS1, we pursue the goal of maximizing the wavefront size without reverting to multi-dimensional tiling. Let $Z$ be the private L2 cache size of each thread, $W_{\max}$ be the size of the largest domain dimension which is traversed, and $W_{\max2}$ be the second largest which is tiled. Let $B_Z$ be the width of the single-form diamond, then $B_Z^2/(2s)$ is its area,

and $B_Z$ can be computed as

$$B_Z \quad := \quad \left\lfloor (2sZW_{\max}W_{\max2}/\left(C_SN\right))^{\frac{1}{2}} \right\rfloor . \tag{4.2}$$

This value determines how many diamonds will fit side by side along the tiling dimension. As we consider large domains, we have sufficiently many independent diamonds to occupy multiple threads. Should this not be the case because of a small tiling dimension, then we can swap the traversal and tiling dimensions or switch to CATS1 which will tile and traverse the same dimension.

Orozco and Gao [44] process their diamonds in rows with a global synchronization between rows, but this is not necessary as Figure 4.3 shows. Because the computation in each diamond depends only on the two diamonds below it, the processing can be easily parallelized irrespective of how many diamonds reside in a row. Moreover, we do not need a global synchronization among threads, instead every diamond simply waits on the two diamonds below it before it starts processing, see the dia-loop in Alg. 4. The a-priori thread to tile assignment may still lead to some idle time, but this is much smaller than Figure 4.3 suggests at first, e.g., the thread that computes the tiny triangle at the right border continues *immediately* with the third diamond in the second row because the two green diamonds below have already finished.

In the previous section, we have seen that CATS1 runs into problems on large 3D domains. CATS2 has no problems in 3D because the size of the wavefront inside the diamond tube that needs to reside in the cache is now further restricted by $B_Z$. Only on enormous 3D or higher dimensional domains, that do not fit into a typical main memory size of 8 GiB, we would need to switch to higher order CATS schemes that are discussed next.

### 4.3.4 Multiple Skewing - General CATS Scheme

By adding more tiling dimensions we can define CATS3, CATS4, etc. In these schemes we still have one traversal dimension but multiple tiling dimensions. The additional complexity in comparison to CATS2 is the more complicated form of space-time tiles, which corresponds to more loops with variable bounds in the algorithm. But even if enormous domain sizes force us to tile multiple dimensions in CATS3 and higher, in contrast to classical multi-dimensional tiling approaches, we tile two dimensions less, one is reserved for the wavefront traversal, the other for vectorization.

When tiling multiple dimensions, we can freely choose which of them should also be parallelized. The tiled and parallelized dimensions use the diamond shape, whereas the tiled-only dimensions may also use space dependent tiles like the parallelograms. On multi-core processors it is sufficient to parallelize just one of the tiling dimensions. Only when

extracting hundredfold parallelism on many-core processors, we would also parallelize more tiling dimensions.

In general, a $d$-dimensional domain admits the use of the CATS$k$ scheme with $k = 1, \ldots, d$. The difference $d - k$ specifies how many dimensions have not been skewed and thus how many inner loops with fixed bounds that scheme has. All values traversed in these loops must reside in the cache, and therefore this difference is usually 0, 1 or 2. If $d - k = 0$ then the cache size poses no problem at all, but the execution of the innermost loop is less efficient because of the variable loop bounds. For common cache sizes of 128KiB–4MiB per core and domain sizes $(100\text{–}1000)^d$, choosing CATS$(d-1)$ for a $d$-dimensional spatial domain is a safe choice that gives fixed loop bounds for the unit stride dimension. We define the general CATS scheme to be this combination of the CATS$k$ schemes. We only deviate in two cases: for 1D problems CATS0 is equivalent to the naive scheme so CATS1 is the better choice; for very large dimension sizes, e.g., $10000^2$ CATS1 would hold the values from the inner loop only for very few timesteps simultaneously and then switching to CATS2 despite the variable loop bounds is better. As a rule of thumb, we switch from CATS$(k-1)$ to CATS$k$ when the wavefront in CATS$(k-1)$ would extend over less than 10 timesteps.

## 4.4 Results

### 4.4.1 Experimental Setup

We compare the performance of the following schemes on iterative stencil computations:

- **NaiveSSE:** Our own hand-parallelized (pthreads [42]) and vectorized (SSE2) naive stencil scheme as described in Section 4.3.1.
- **PluTo [7]:** Code transformed by the automatic parallelizer and locality optimizer for multicores PluTo, version 0.4.2.
- **PeakDP:** The measured computational peak in double precision. We obtain this value by performing a sequence of independent multiply-add operations in registers. PeakDP models the absolute upper bound for any computation on a machine. The ultimate goal of optimized stencil computations is to achieve a high fraction of this peak as no optimization of stencil codes will reach this value because of the dependency between the stencil operations.
- **CATS:** Our general cache accurate time skewing scheme with the selection of individual schemes described in Section 4.3.4. The innermost loop uses a vectorized (SSE2) kernel and parallelization uses pthreads.

**Table 4.1.** Hardware configurations of our test machines. The machines have been chosen such that one (Opteron) has a modest ratio between measured system and cache bandwidth, while the other (Xeon) has a high ratio. This ratio is the main source of acceleration of time skewing against naive schemes.
The measured bandwidth numbers have been obtained with the RAMspeed benchmarking tool and the double precision (DP) FLOPS numbers come from our own SSE benchmarks. For the peak DP number we perform independent multiply-add operations on registers, for the stencil DP number we run the inner stencil computation (products and accumulation) on registers. This value is lower because of the read-after-write dependencies in the computation. All benchmarks show results for the entire machine achieved with 4 threads.

| Brand | AMD | Intel |
|---|---|---|
| Processor | Opteron 2218 | Xeon X5482 |
| Code-named | Santa Rosa | Harpertown |
| Frequency | 2.6 GHz | 3.2 GHz |
| Number of sockets | 2 | 1 |
| Cores per socket | 2 | 4 |
| L1 Cache per core | 64 KiB | 32 KiB |
| L2 Cache per core | 1 MiB | 3 MiB |
| Operating system | Linux 64 bit | Linux 64 bit |
| Parallelization | 4 pthreads | 4 pthreads |
| Vectorization | SSE2 | SSE2 |
| Compiler | g++ 4.3.2 | icpc 11.1 |
| Measured L1 Bandwidth | 79.3 GB/s | 194.6 GB/s |
| Measured L2 Bandwidth | 40.6 GB/s | 64.2 GB/s |
| Measured Sys. Bandwidth | 11.2 GB/s | 6.20 GB/s |
| Measured Peak DP FLOPS | 20.8 G | 40.8 G |
| L2 Band./Sys. Bandwidth | 3.6 | 10.4 |
| Peak DP/(Sys. Band./8B) Balanced arith. intensity for Sys. | 14.9 | 52.6 |

Our hardware configuration is listed in Table 4.1. As general compiler options we use -O3 -funroll-loops and for the icpc compiler also -xHOST -no-prec-div. The NaiveSSE scheme does not require any parameters, it only needs a scalar and a vectorized kernel that are called from the nested loops.

For PluTo-0.4.2 we use -tile -l2tile to tile the code for the L1 and L2 cache, -multipipe to extract multiple degrees of parallelism, -parallel to parallelize the code using OpenMP, -unroll to automatically unroll up to two loops, and -nofuse to separate all strongly-connected components in the dependence graphs. The options -unroll -nonuse do not make a difference in performance in our tests. In 3D, we decided to omit the option -l2tile as the transformation process was taking hours and did not provide performance gains. We use the original examples provided with PluTo and modify them from constant to variable stencil where necessary. It is not feasible to hand-vectorize the transformed code because of the high number of generated loops, e.g., 142 loops for the constant 7-point stencil in 3D. However,

we ensure the best possible performance by retransforming and recompiling the examples every time with compile-time known domain sizes and aggressive icpc auto-vectorization, the compilation process alone takes about 15 minutes.

CATS takes as parameters the size of the last cache level (L2 for us), the order of the stencil $s$, the memory size of a data type and optionally additional cache requirements, e.g., the matrix coefficients. CATS is implemented as a library not a code generation framework. The kernel may perform arbitrary index calculations and non-linear operations on the data within the stencil region $\{-s, \ldots, +s\}^d$ and on the specified amount of additional values like matrix coefficients. Beside the parameters, the user only provides a scalar and a vectorized version of the kernel, the same kernels used by the optimized naive scheme.

Our test applications comprise constant and variable stencils in 2D and 3D with 0.5 to 128 million double precision elements. In 2D, we have squares ranging from $706^2$ to $11282^2$ elements and in 3D, cubes from $80^3$ to $500^3$. In case of constant stencils, this amounts to a memory consumption of up to 2GiB for the two vectors, and in case of variable stencils we use at most 32 million elements consuming 0.5GiB plus 1.75GiB for the matrix in 3D. We use a general 5-point stencil in 2D (5 muls plus 4 adds equal 9 flops) and a 7-point in 3D (7 muls plus 6 adds equal 13 flops). The number of iterations is either $T = 100$ (solid graphs in the figures), or $T = 10$ (dashed graphs in the figures). The last stencil application is the FDTD 2D example (11 flops) that comes with PluTo.

All figures show the execution time in seconds against the number of elements in millions with both axes being logarithmic. The number of elements doubles between two consecutive graph points, but the doubling is not totally exact because of the square or cubic root operations involved in computing a square or cube with a predefined number of elements.

### 4.4.2 Constant Stencil

In this section, we present results for constant stencils of order $s = 1$. Figures 4.5 and 4.6 show the execution times for 2D spatial domains and Figures 4.7 and 4.8 for 3D. From the graphs, we can draw the following common conclusions:

- **Large slowdown of the naive scheme on the Xeon when transitioning from 0.5 to 1.0 million elements.**
  The Xeon has 12MiB of L2 cache (cf. Table 4.1), so that two vectors of 0.5 million elements ($2 \cdot 8\text{B} \cdot 0.5\text{M} = 8\text{MB}$) fit into the cache. The one million elements case already requires 16MB, which exceed the cache size, so the performance of the naive scheme suffers a large slowdown and from thereon becomes completely limited by the available system bandwidth. The CATS scheme, on the other hand, has a more consistent scaling and simply ignores the fact that the data does not fit into the

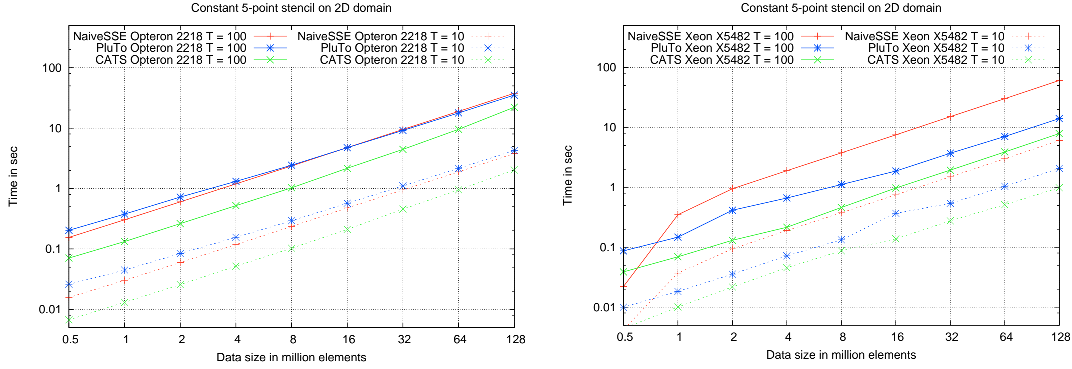



**Figure 4.5.** Timings of the Opteron 2218 with constant stencils in 2D. GFLOPS for 128 million elements with $T = 100$: NaiveSSE Opteron 3.4, PluTo Opteron 3.6, CATS Opteron 5.8 (28% of PeakDP).

**Figure 4.6.** Timings of the Xeon X5482 with constant stencils in 2D. GFLOPS for 128 million elements with $T = 100$: NaiveSSE Xeon 1.9, PluTo Xeon 8.2, CATS Xeon 16.2 (40% of PeakDP).

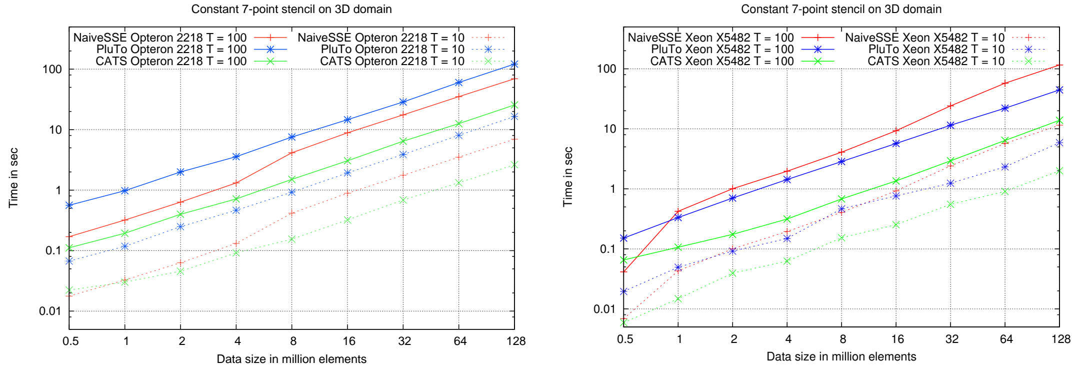



**Figure 4.7.** Timings of the Opteron 2218 with constant stencils in 3D. GFLOPS for 128 million elements with $T = 100$: NaiveSSE Opteron 2.4, PluTo Opteron 1.5, CATS Opteron 6.4 (31% of PeakDP).

**Figure 4.8.** Timings of the Xeon X5482 with constant stencils in 3D. GFLOPS for 128 million elements with $T = 100$: NaiveSSE Xeon 1.4, PluTo Xeon 3.7, CATS Xeon 13 (32% of PeakDP).

cache any more. This causes the CATS graph for $T = 100$ iterations on the Xeon in 2D (Figure 4.6) and 3D (Figure 4.8) to come close to the naive graph for $T = 10$ iterations on large problems. The PluTo scheme also scales consistently but at a much lower level. The Opteron does not show the jump on the naive scheme because its 4MiB of L2 cache can not accommodate two copies of the 0.5 million elements, so it is already in the slow mode determined by the system bandwidth.

- **The Opteron is faster than the Xeon on the naive scheme but slower on PluTo and CATS.**
  The faster execution on the naive schemes is directly related to the higher system bandwith on this machine as it is the limiting performance factor, see Table 4.1. For the time skewing PluTo and CATS schemes, on the other hand, the system bandwidth is less relevant even when the data size exceeds the cache size more than hundredfold, as in the case of the 128 million element examples with 1GiB of data

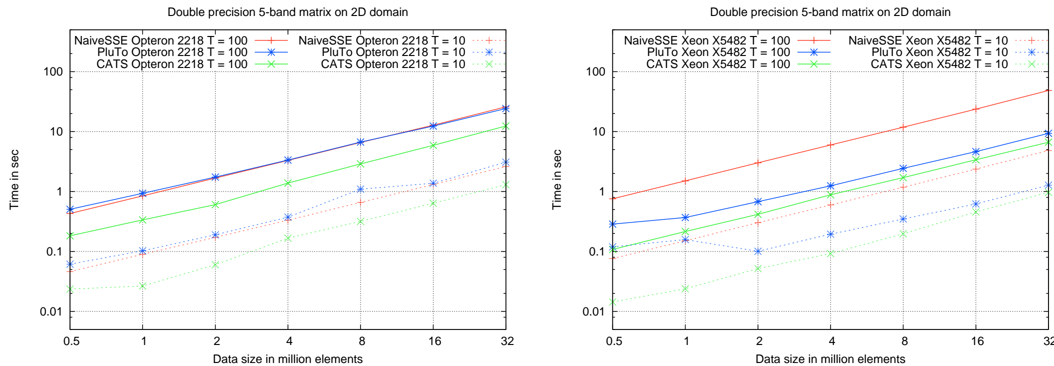



**Figure 4.9.** Timings of the Opteron 2218 with a banded matrix in 2D. GFLOPS for 32 million elements with $T = 100$: NaiveSSE Opteron 1.1, PluTo Opteron 1.2, CATS Opteron 2.8 (13% of PeakDP).

**Figure 4.10.** Timings of the Xeon X5482 with a banded matrix in 2D. GFLOPS for 32 million elements with $T = 100$: NaiveSSE Xeon 0.6, PluTo Xeon 3.1, CATS Xeon 4.9 (12% of PeakDP).

for each vector. The cache bandwidth is the decisive factor, hence the Xeon is better and consequently shows better results despite its low system bandwidth.

For the achievable acceleration factor the ratio of cache to system bandwidth (3.6 Opteron, 10.4 Xeon, see Table 4.1) and the scheme's ability to exploit this ratio are important. CATS exploits this ratio well outperforming the naive scheme on the Opteron by a factor 2 on average, and on the Xeon by at least 7.5x. PluTo does also benefit from the ratio but to a smaller extent. It performs on average slower than the naive scheme on the Opteron, but faster on the Xeon due to the bigger ratio on the Xeon.

- **Performance in 2D is generally better than in 3D.**
  This is not surprising as the surface area to volume ratio is worse in 3D but the effect on the schemes varies substantially. The naive scheme in 3D maintains the same performance as in 2D on smaller domains, which makes sense because the same amount of data is transported and system bandwidth is the limiting factor. Beyond a certain size in 3D, four 2D slices (3 input plus 1 output) of the domain do not fit into the cache anymore so that stencil neighbors have to be brought into cache multiple times and performance degrades. PluTo works best in 2D where it is on par with the naive scheme on the Opteron and much faster on the Xeon. In 3D the performance degrades by more than 2x in both cases. CATS also slows down in 3D but only by around 20%, so the speedup over PluTo grows to more than 3.5x.

### 4.4.3 Banded Matrix

If the stencil is not constant but rather varies across the domain, then its application corresponds to a banded matrix vector product. In Section 5.3 we assumed $N_S$ as the

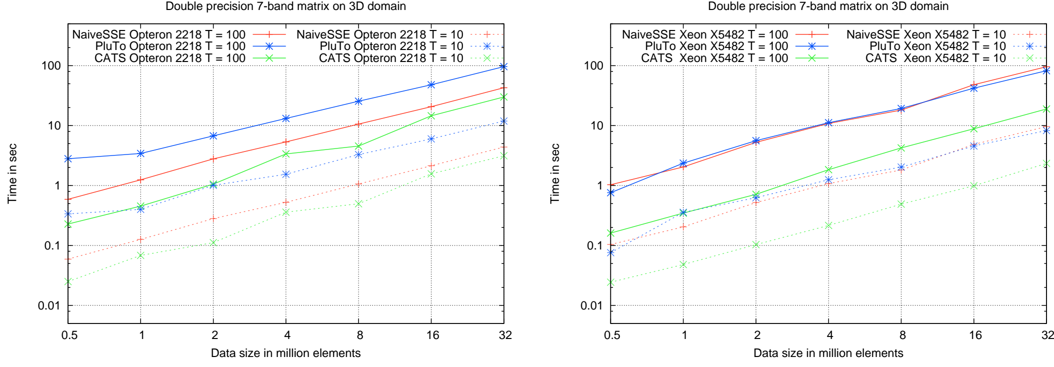



**Figure 4.11.** Timings of the Opteron 2218 with a banded matrix in 3D. GFLOPS for 32 million elements with $T = 100$: NaiveSSE Opteron 1.0, PluTo Opteron 0.4, CATS Opteron 1.5 (7% of PeakDP).

**Figure 4.12.** Timings of the Xeon X5482 with a banded matrix in 3D. GFLOPS for 32 million elements with $T = 100$: NaiveSSE Xeon 0.4, PluTo Xeon 0.5, CATS Xeon 2.5 (6% of PeakDP).

number of non-empty stencil elements, this corresponds to the number of bands in the matrix. For the space-time traversal this means that not only the vector components (domain values) must reside in the cache but also the corresponding matrix entries. We need the matrix entries only for the current wavefront during the computation, so $C_S$ must be replaced by $C_S + N_S$ in our formulas Eq. 4.1 and Eq. 4.2 that compute the maximum extent of the wavefront. We run performance tests with $T = 10$ and $T = 100$ iterations shown in Figures 4.9 and 4.10 for 2D and Figures 4.11 and 4.12 for 3D. We make similar observations to the constant stencil case.

- The Opteron is faster than the Xeon on the naive scheme but slower on PluTo and CATS. The main reason is the same as for the constant stencil: for the naive scheme the system bandwidth matters most while for the time skewing schemes the cache bandwidth is more important. However, the performance ratios between the Opteron and the Xeon for the naive scheme are now larger and for PluTo and CATS smaller than before, because the additional matrix transfers increases the influence of the system bandwidth speed on all schemes.

- Performance in 2D is generally better than in 3D. This effect is further enforced by the fact that the 2D matrix has $N_S = 5$ bands while the 3D matrix has $N_S = 7$. This time the naive scheme is the least affected by the transition from 2D to 3D. Therefore, CATS's advantage over the naive scheme drops from 2.5x to 1.5x on the Opteron and from 8.2x to 6.2x on the Xeon. For PluTo it means that equal performance with the naive scheme drops to worse on the Opteron and much better performance drops to equal on the Xeon.

### 4.4.4 Scalability

| GFLOPS of ... | 1 thread | 2 threads | 4 threads |
|---|---|---|---|
| CATS Opteron | 1.7 | 3.3 | 6.4 |
| CATS Xeon | 5 | 9.6 | 13 |

The table above shows how CATS scales from one to four threads on the constant 7-point stencil for the 128 million elements problem in 3D with $T = 100$ iterations. Although this is a memory-bound problem, both the Opteron and the Xeon scale almost perfectly from one to two threads. Supported by higher system bandwidth (11.2 GB/s) the Opteron also scales well to four threads, while the lower system bandwidth (6.20 GB/s) of the Xeon limits the gains from additional cores.

### 4.4.5 High Order Stencils

| GFLOPS of ... | $s = 1$ | $s = 2$ | $s = 3$ |
|---|---|---|---|
| NaiveSSE Opteron | 2.4 | 3.1 | 3.1 |
| PluTo Opteron | 1.5 | 0.9 | 0.9 |
| CATS Opteron | 6.4 | 7.5 | 4.7 |
| GFLOPS of ... | $s = 1$ | $s = 2$ | $s = 3$ |
| NaiveSSE Xeon | 1.4 | 1.9 | 1.7 |
| PluTo Xeon | 3.7 | 4.3 | 1.9 |
| CATS Xeon | 13.0 | 8.5 | 4.6 |

Up to now we have shown results for the most common stencils of order $s = 1$. Stencils with larger orders worsen the surface area to volume ratio of the space-time tiles. Above we compare the performance of the constant 7-point stencil of order, the 13-point stencil of order 2, and the 19-point stencil of order 3 for the 128 million elements problem in 3D with $T = 100$ iterations. We see that CATS maintains a clear advantage in all cases despite the different performance dependence of the schemes on the order $s$.

### 4.4.6 Application: FDTD Solver

The previous sections analyzed basic stencil computations on a scalar domain with constant or variable weights in detail. In practice, these basic stencil computations appear in different variations. In this section we examine one such variation that is often used to demonstrate the efficiency of time skewing schemes, namely a 2D Finite Difference Time Domain (FDTD) electromagnetic kernel. This kernel is basically used to solve the discretized Maxwell's equations numerically [61].

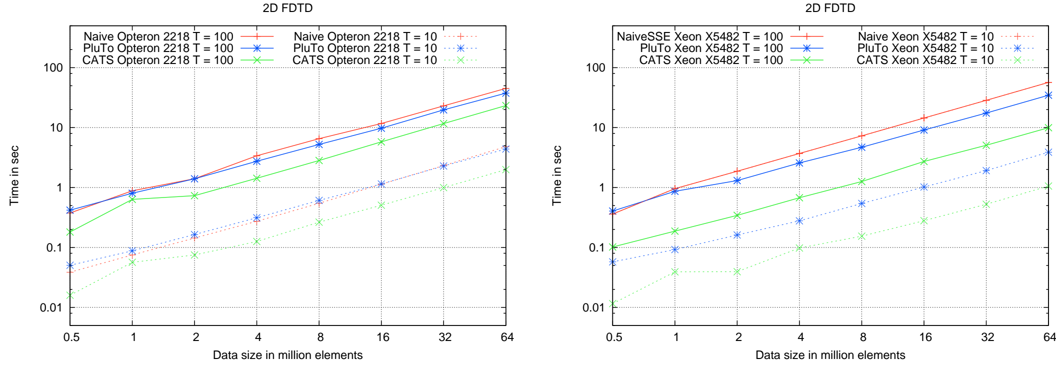



**Figure 4.13.** Timings of the Opteron 2218 for FDTD in 2D. GFLOPS for 64 million elements with $T = 100$: NaiveSSE Opteron 1.6, PluTo Opteron 1.9, CATS Opteron 2.7 .

**Figure 4.14.** Timings of the Xeon X5482 for FDTD in 2D. GFLOPS for 64 million elements with $T = 100$: NaiveSSE Xeon 1.2, PluTo Xeon 2.0, CATS Xeon 6.4 .

For PluTo, we use the code given in the paper [7], which is also included as a software example. For CATS we fuse the three loops in 2D FDTD manually to obtain a single kernel. Then we write a vectorized version of this kernel and pass its pointer to the naive scheme and CATS. Figures. 4.13 and 4.14 show the results. Because this is a vector valued problem with 3 doubles for each point in the space-time, more data must be kept in cache which forces the wavefronts to become smaller. Not surprisingly the results are a slowed down version of the 2D constant stencil tests in Figures. 4.5 and 4.6. PluTo has a small advantage over the naive implementation of around 1.2x on the Opteron and a clearer advantage of 1.7x on the Xeon. CATS beats the naive scheme by 1.7x (1.4x vs. PluTo) on the Opteron and 5.3x (3.2x vs. PluTo) on the Xeon.

## 4.5 Conclusion

We have presented CATS, a cache accurate time skewing scheme for iterative stencil computations on multi-core processors. It is based on a novel usage of a wavefront traversal in multi-dimensional time skewing, an unconventional departure from the complexity of the commonly used techniques of multi-dimensional tiling and multi-level tiling. The strategy is particularly successful on constant stencils of order 1, where the algorithm breaks the dependence on the low system bandwidth and achieves a high fraction of the computational peak in 2D and 3D even when operating on gigabyte large domains. This is a significant improvement over the optimized naive scheme and the state-of-art in automatic optimization. For large stencils and banded matrices the system bandwidth limits the performance again but in comparison CATS maintains a clear advantage.

We expand the state-of-art by proposing a scheme that delivers high performance stencil computations via data locality, regular memory access, and vectorization optimizations.

Previous schemes recourse to complex techniques such as multi-dimensional and multi-level tiling to optimize the stencil codes on multi-dimensional domains whereas CATS shows how to obtain better performance results without using any of these complicated techniques.

# Chapter 5

# Performance Modelling

In this Chapter, we analyze in detail the impact of the system and cache bandwidths on efficient stencil computations. While the naive implementation is known to be memory bound and to scale linearly with the system bandwidth, for the time skewing methods the situation is quite different because cache misses are reduced to such great extent that the cache bandwidth becomes an important performance factor. For this more general situation we develop a performance model, validate it across many processor generations and thus determine how scaling the system and cache bandwidths influences hardware performance [57]. These insights are useful for identifying the most performance relevant features of future systems with respect to stencil computations.

The next section discusses our hardware and software setup for the performance and cache analysis. Section 5.3 compares the execution times of the naive and the CATS scheme (Chapter 4) for varying problem sizes. In Section 5.4 we vary the cache size and explain how CATS can predict the performance on virtual machines with different cache sizes. Based on the previous data, the performance model is developed in Section 5.5 and Section 5.6 validates it and estimates the performance for new hardware configurations.

## 5.1 Hardware Setup

Table 5.1 lists the configuration of our hardware. We refer to the machines by the processor name throughout the paper. The first two represent the older generation of single-core, multi-CPU workstations. The last two offer four cores either in one or two sockets and feature integrated memory controllers. The Core i7 940 is also used to simulate a Core i5 dual-core system by executing only two threads on this processor and this configuration is labeled Core i5 Sim.

**Table 5.1.** Hardware configurations of our machines. The first half of the table refers to the technical specification of the CPUs. The second half presents results of synthetic benchmarks on these machines.

| Brand | Intel | AMD | Intel | AMD | Intel |
|---|---|---|---|---|---|
| Processor | Xeon MP | Opteron 250 | Core i5 Sim | Opteron 2218 | Core i7 940 |
| Code-named | Gallatin | Troy | - | Santa Rosa | Bloomfield |
| Frequency | 3.06 GHz | 2.4 GHz | 2.93 GHz | 2.6 GHz | 2.93 Hz |
| Number of sockets | 2 | 2 | 1 | 2 | 1 |
| Cores per socket | 1 | 1 | 2 | 2 | 4 |
| L1 Cache per core | 8 KiB | 64 KiB | 32 KiB | 64 KiB | 32 KiB |
| L2 Cache per core | 512 KiB | 1024 KiB | 256 KiB | 1024 KiB | 256 KiB |
| L3 Cache per core | 1024 KiB | - | 4096 KiB | - | 2048 KiB |
| Number of threads | 2 | 2 | 2 | 4 | 4 |
| Measured L1 Bandwidth | 44.4 GB/s | 36.4 GB/s | 91.1 GB/s | 79.3 GB/s | 182.3 GB/s |
| Measured L2 Bandwidth | 24.3 GB/s | 21.0 GB/s | 61.3 GB/s | 40.6 GB/s | 124.0 GB/s |
| Measured L3 Bandwidth | 17.3 GB/s | - | 44.5 GB/s | - | 88.2 GB/s |
| Measured Sys. Bandwidth | 3.2 GB/s | 6.4 GB/s | 14.4 GB/s | 11.2 GB/s | 17.8 GB/s |
| Last Level Cache/Sys. Band. | 5.4 | 3.3 | 3.3 | 3.6 | 3.6 |
| Measured Peak DP FLOPS | 11.5 G | 9.6 G | 19.6 G | 20.1 G | 39.1 G |

The second half of the table presents results of synthetic benchmarks. The 'number of threads' row shows how many threads were used during the computation. This number is fixed on each machine. The bandwidth benchmarks have been performed with the RAMspeed benchmarking tool with SSE reads. All x86-64 capable machines run 64-bit Linux and the GNU C++ 4.3 compiler. On the Xeon MP machine we use 32-bit Linux and the GNU C++ 4.2.1 compiler.

The measured system floating point performance numbers come from our own benchmarks. The peak performance value (see Table 5.1) is the maximum number of independent alternating multiply and add instructions executed per second on all available SSE units. This gives us the maximum overall system performance. However, as our performance model assumes faster computation than data fetching from the cache (the problem is cache bandwidth bound in the cache), we are more interested in the application specific performance of stencil computations. Therefore, we implement the 7-point stencil computation in registers as a series of accumulations of products and present the results as measured stencil double precision (DP) FLOPS in Table 5.1. This value is lower than the measured peak performance because of the dependency (read-after-write) between the instructions in the pipeline.

## 5.2 Software Setup

We use two schemes for the tests with iterative stencil computations. The naive scheme consists of perfectly nested for-loops that traverse the entire spatial domain and the outer most loop that repeats the stencil application multiple times (NaiveSSE_3D() in Algo-

rithm 5). The cache accurate time skewing (CATS) scheme exploits temporal locality between the consecutive iterations of the stencil. It also consists of nested for-loops and allows similar parallelization and vectorization as the naive scheme, only there are more loops and they appear in different order(CATS_3D() in Algorithm 5). Both schemes are vectorized with SSE2 intrinsics on the inner most loop and parallelized using pthreads on the outer most loop using the number of threads according to Table 5.1.

**Algorithm 5** Cache accurate time skewing in 3D. Only few transformations are necessary to obtain much faster parallel C++ code from the naive implementation. Function stencil_SSE() contains the stencil computation vectorized along the x-axis from 0 to WIDTH. The tile sizes are chosen such that they fit into the last level cache, more details are given in [58].

```
void NaiveSSE_3D ()
{
  for(int t = 0; t < T; t++) {
    for(int z = 0; z < DEPTH; z++) {
      for(int y = 0; y < HEIGHT; y++) {
        stencil_SSE(t, z, y, 0, WIDTH);
      }//y
    }//z
  }//t
}

void CATS_3D (int threadID)
{
  for( TileIt tile = tileSet[threadID].begin();
      tile != tileSet[threadID].end(); ++tile) {
  wait_on_dependencies(tile);

  for(int z = 0; z < DEPTH; z++) {
    for(int t = tstart(tile,z); t < tend(tile,z); t++) {
    for(int y = ystart(tile,z,t); y < yend(tile,z,t); y++) {
      stencil_SSE(t, z-s*t, y, 0, WIDTH);
    }}//t,y
  }//z

  }//tile
}
```

All our performance tests share certain properties:

- computation in double precision,
- ping-pong iterations with two vectors,
- constant general 7-point stencil in 3D (7 multiplications plus 6 additions) with Dirichlet boundary condition,
- 3D domains ranging from 0.5 to 128 million elements, corresponding to 8MiB-2GiB of data.

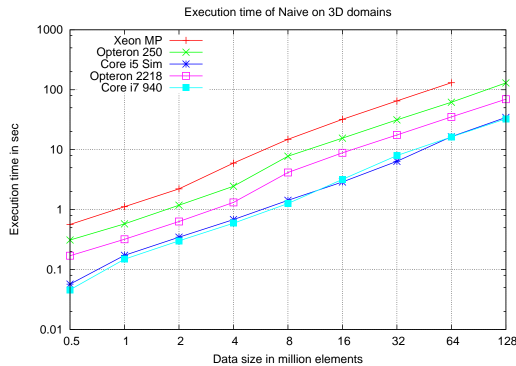



Figure 5.1. Execution time of the naive scheme for varying domain sizes in 3D.

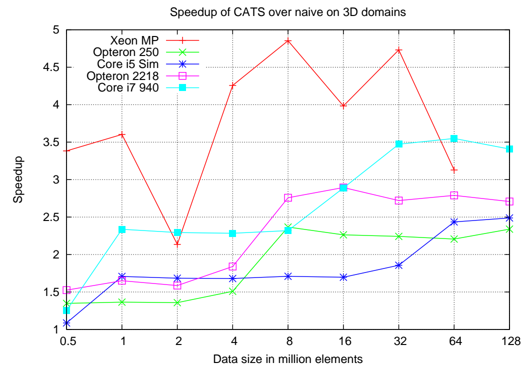



Figure 5.2. Speedup of the CATS scheme against the naive scheme for varying domain sizes in 3D.

We have also tested some other configurations with single precision, variable stencils (banded matrix), in-place stencil updates, and 1D and 2D domains; however, we have obtained qualitatively similar relations as discussed below although the quantitative results can vary significantly with the parameters. In the following, we prefer to deliver a consistent analysis from start to end for the specified parameters rather than jumping between different parameter configurations.

In Section 5.4, we simulate cache misses of the naive and CATS scheme. The cache miss analysis is performed using the cachegrind profiler from the valgrind 3.2.1 tool suite. We simulate a processor with one cache level and interpret recorded read and write misses as the misses of the last cache level of our machines. Because cachegrind does not simulate multi-threaded programs realistically, we record the cache misses separately for each thread on its piece of the domain and sum up the values. This can lead to slightly higher values because of some additional data reuse in the shared L3 cache, but Section 5.4 shows that for CATS this is not a problem because there is hardly any additional data reuse apart from the data explicitly accounted for reuse.

## 5.3 Naive and Time Skewed Stencil Computations

The naive scheme implementation progresses with the entire domain one timestep after the other. As the domain size is usually bigger than the cache, each pass thrashes the cache contents entirely. This makes the naive scheme depend mainly on the system bandwidth for performance. Figure 5.1 shows the linear relation between the domain size and execution time. The only noticeable non-linearity can be observed for the Core machines at the transition from 0.5 to 1 million elements. This jump is caused by the large L3 cache size when two 0.5 million vectors fit completely into the cache, but the two 1.0 million vectors do not.

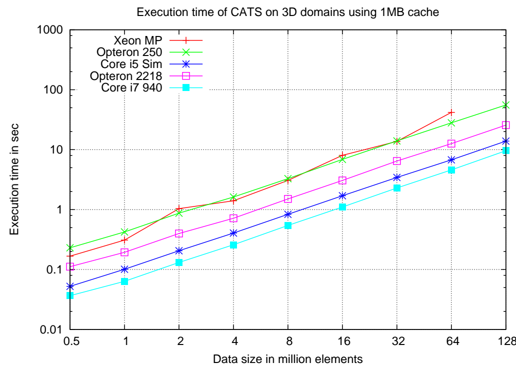



**Figure 5.3.** Execution time of the CATS scheme for varying domain sizes in 3D with cache size parameter 1024KiB.

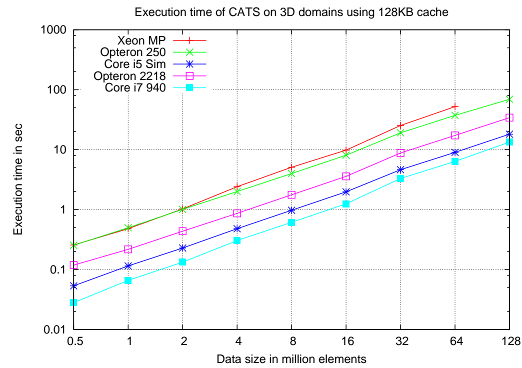



**Figure 5.4.** Execution time of the CATS scheme for varying domain sizes in 3D with cache size parameter 128KiB.

Figure 5.2 shows the speedup of the CATS scheme over the naive implementation. For domains with moderate size, the naive implementation benefits from the cache capacity and reuses data elements from neighboring 2D slices. But the speedup increases in a step when neighboring 2D slices do not fit into the cache any more. As the Core i5 Sim machine runs with 4MiB of cache memory per thread, this increase happens only at 64 million elements, which is much later than in the case of any other platform.

The maximum speedup value is closely related to the ratio of the last cache level bandwidth to the system bandwidth (see Table 5.1). Systems with larger discrepancy between bandwidths benefit more from the increased temporal locality of the computation in CATS and effectively generate larger speedups. An example of such a system is the Xeon MP which has the largest ratio of bandwidths namely 5.4 (Table 5.1). Accordingly, for 32 million elements, CATS achieves a speedup of almost 5 times. For the same domain size, the Core i7 940 with a bandwidth ratio of 3.6 accelerates by a 3.5 factor. A similar result is observed for the Opteron 2218 machine. The smaller speedups on the last two systems are not surprising because of their integrated memory controllers and the dual/triple channel memory interfaces providing up to 6 times more memory bandwidth than the Xeon MP.

Figures 5.3 and 5.4 show the execution times of the CATS scheme. The algorithm requires prior knowledge about the available cache size per thread. We use 1024KiB and 128KiB for the cache size settings, respectively. For both configurations CATS shows consistently faster execution time than the naive scheme. The previously noticed non-linear performance scaling at the transition from 0.5 to 1 million elements on the Core machines is no longer visible. The CATS scheme scales consistently with the domain size no matter if the entire domain fits into the cache or not. Clearly, in the case of the 128KiB cache size, CATS exploits temporal localities less efficiently compared to the 1MiB setting, resulting in increased execution time. But the 128KiB configuration of CATS is still noticeably faster than the naive implementation, e.g. for 32 million elements, the speedup is 2.6x on

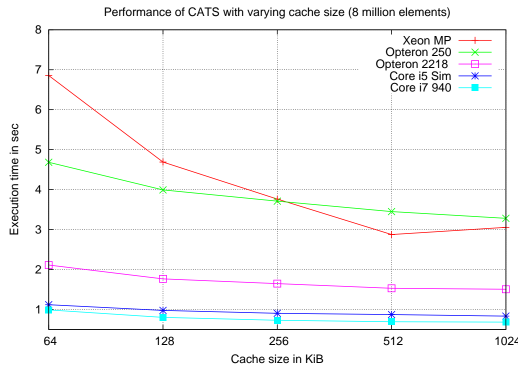



**Figure 5.5.** Execution time of CATS for a $200^3$ domain and varying cache sizes.

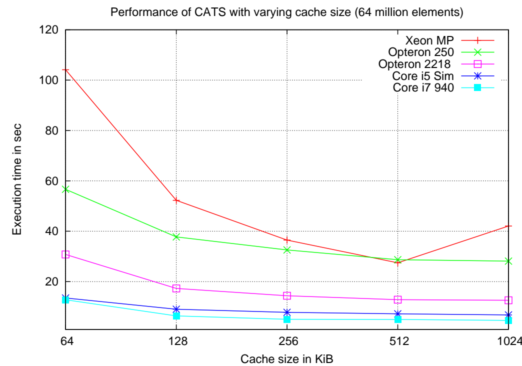



**Figure 5.6.** Execution time of CATS for a $400^3$ domain and varying cache sizes.

the Xeon MP compared to 4.7x for the 1MiB cache setting. On machines with smaller cache to system memory bandwidth ratio like Core i5 or Opteron 2218, the speedup is less impressive, but the execution times are still half that of the naive approach.

For the naive scheme the Opteron 250 is faster than Xeon MP (see Figure 5.1) by a factor of 1.8x. As the algorithm is memory bound, the difference in performance directly relates to the difference in the measured system bandwidths: 6.4 GB/s for the Opteron 250 vs. 3.2 GB/s for the Xeon MP (see Table 5.1). However, for the CATS scheme, cache bandwidth is the main performance limiting factor. For the Opteron 250 and the Xeon MP, the cache bandwidth is approximately equal and therefore the CATS scheme performs similarly on both machines.

A comparison of the Core systems (see Figures 5.3 and 5.4 ) leads to similar conclusions. Both systems are equal in terms of system bandwidth, effectively achieving the same results for the naive approach. However, for the CATS scheme the quad-core Core i7 940 performs significantly better. The difference comes from the increased aggregated cache bandwidth when all four cores are in use. In general, one can say that the performance order of the machines in Figure 5.1 reflects the ranking in system bandwidth, whereby their order in Figure 5.3 reflects the ranking in cache bandwidth.

## 5.4 Varying Cache Size

The previous section has shown two different plots of CATS performance depending on the passed cache size parameter: Figure 5.3 for $Z = 1024$KiB and Figure 5.4 for $Z = 128$KiB. In this section we want to fix the problem size to either a $200^3$ domain (8 million elements) or a $400^3$ domain (64 million elements) and look at the performance scaling on machines with different cache sizes. Figures 5.5 and 5.6 show the execution times for varying cache sizes. This will be used in the next section to derive a performance model. But first

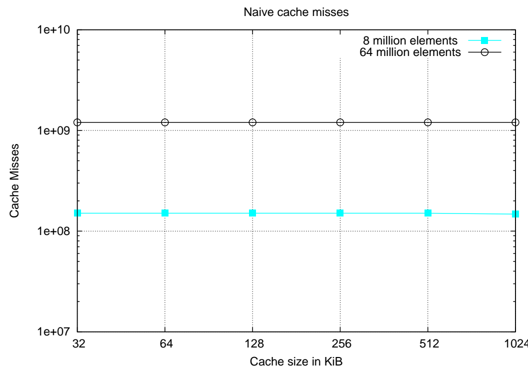



**Figure 5.7.** Cache misses of the naive scheme for a $200^3$ and a $400^3$ domain and varying cache sizes.

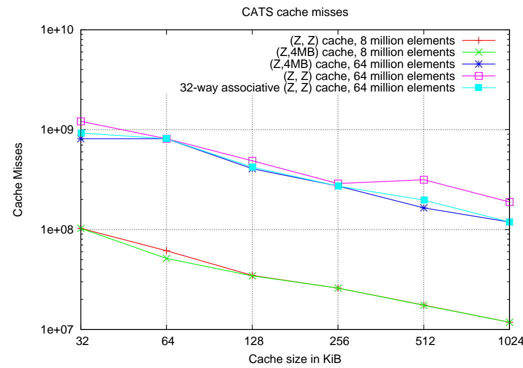



**Figure 5.8.** Cache misses of the CATS scheme for a $200^3$ and a $400^3$ domain and varying cache sizes.

we need to explain why these numbers predict the performance of CATS on hardware configurations with smaller cache sizes, although the actual machines on which we execute have obviously a fixed hardware cache size.

The reason is an invariance property of the CATS scheme. Given a cache size parameter $Z$, it will incur the same number of cache misses on a machine with $Z$ cache, $2 \cdot Z$ or even $4 \cdot Z$ cache. It only matters if the actually available cache size is bigger than or equal to the specified parameter. The CATS scheme optimizes the entire computation very carefully with respect to the given cache size parameter, so even if the actual cache is bigger, there will be hardly any additional savings on cache misses.

Figure 5.8 confirms the above reasoning. If CATS is fed with the same $Z$ value, then the cache misses on a machine with 4MiB cache size (CATS $(Z, 4\text{MiB})$) are only insignificantly lower than on a machine with $Z$ cache (CATS $(Z, Z)$). Only in case of the large $400^3$ domain (128 million elements), we obtain a discrepancy due to the imperfect cache associativity. Increasing the cache associativity from 8-way to 32-way recovers the almost identical behavior.

This invariance of CATS is very useful, because the arithmetic intensity of stencil computations is very low, so their performance depends mainly on the number of cache misses and the system and cache bandwidths of the machine. So if the cache misses do not change when we run on a machine with much larger cache than the cache size parameter, then the performance should not change either. Practically, by setting the cache size parameter $Z$ to some value, e.g. 128KiB, we obtain the execution time of a virtual machine with this cache size $Z = 128$KiB, even though the actual execution takes place on a machine with 4MiB cache size. Figure 5.8 clearly shows that even such big difference of 4MiB to 128 KiB has almost no impact on the number of incurred cache misses and thus the performance of CATS.

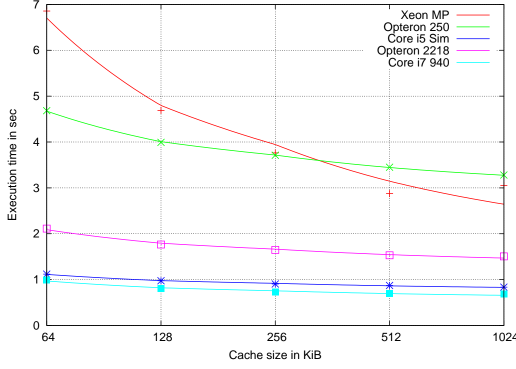



**Figure 5.9.** Execution time of CATS for a $200^3$ domain and varying cache sizes. The points show the measured execution time, while the lines show our fitted performance model based on the number of cache misses.

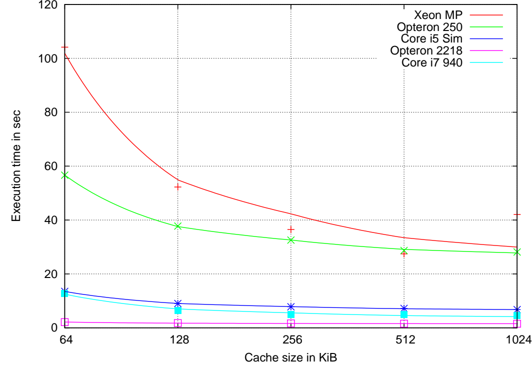



**Figure 5.10.** Execution time of CATS for a $400^3$ domain and varying cache sizes. The points show the measured execution time, while the lines show our fitted performance model based on the number of cache misses.

Something similar holds trivially for the naive scheme as demonstrated in Figure 5.7. In this case the number of cache misses is almost constant no matter how big the cache is, because the entire domain is fetched for each iteration of the stencil, and the entire domain is always larger than the cache size, so no data reuse between the stencil iterations can occur. Comparing Figures 5.7 and 5.8, we see that CATS produces fewer cache misses even for small cache sizes. For growing cache sizes, cache misses incurred by CATS decrease rapidly, producing less than a tenth of the naive cache misses for a 1024KiB cache size.

## 5.5 Performance Model

In the previous section, we have explained how we can use CATS to estimate the execution time on the same machine where we virtually vary the size of the available cache. In this section we derive a performance model that links the number of simulated cache misses directly to the measured execution time. The model estimates the execution time $E(m)$ as

$$E(m) := \; C_l \cdot (m_r(Z) + m_w(Z))/b_{\text{sys}} \; + \; (C_d T N N_s - C_l \cdot m_r(Z))/b_{\text{cache}}, \tag{5.1}$$

where $m_r(Z)$ and $m_w(Z)$ are the simulated numbers of read and write cache misses for varying cache size $Z$, $C_l = 128\text{B}$ is the size of the cache line, $C_d = 8\text{B}$ is the size of a domain element (double precision), $T$ is the number of iterative stencil applications, $N$ is the number of elements in the domain, and $N_s = 7$ is the number of non-zero stencil weights. The model has two free parameters which are the system bandwidth $b_{\text{sys}}$ and the cache bandwidth $b_{\text{cache}}$. The parameters are estimated by a least-square-fitting of the model to the measured execution times from Figures 5.5 and 5.6.

The first addend in Eq. 5.1 contributes with the time necessary for the transfer from main memory due to the cache misses. The second addend corresponds to the time of all remaining transfers (there are $TNN_s$ double computations) from the cache to the processing units. Figures 5.9 and 5.10 show the measured execution times as points and the fitted performance models as line plots.

The model considers only a two level memory hierarchy: the system memory level where the domain resides and a cache level in which the temporal locality is exploited. Real machines have multiple cache levels and so the effect of the temporal locality is higher or smaller depending on which cache level it occurs in. Thus, the model does not capture the secondary effects of higher level caches. However, the cache bandwidth ratios on-chip are clearly smaller than the bandwidth ratios between the last level cache and the system bandwidth (Table 5.1) and the schemes do not use any explicit optimization for high level caches, so that the secondary effects are less relevant.

Previous comparison of the execution time of the naive scheme in Figure 5.1 against CATS in Figure 5.4 reveals that even on a machine with just 128KiB cache, CATS performs clearly better. Looking at the cache misses in Figures 5.7 and 5.8, we see that at 128KiB CATS has already a dramatic reduction of cache misses against the naive implementation. For larger caches the difference in cache misses continues to grow at the same pace, but Figures 5.5 and 5.6 show diminishing performance returns from the cache miss reduction after 128KiB. Why has the cache miss reduction at first a strong impact on performance while later this impact is much smaller? This behavior can be understood from Eq. 5.1 by computing the speedup obtained from halving the cache misses:

$$\begin{aligned} \frac{E(2m)}{E(m)} &= 1 + \frac{C_l \cdot ((m_r + m_w)/b_{\text{sys}} - m_r/b_{\text{cache}})}{E(m)} \geq 1 + \frac{C_l(m_r + m_w)C_b}{E(m)} \qquad (5.2) \\ C_b &:= (1/b_{\text{sys}} - 1/b_{\text{cache}}) > 0\,. \end{aligned}$$

At first we see that the speedup depends directly on the discrepancy between the system and cache bandwidth encoded in $C_b$. There is also a second effect; in the beginning when the data traffic produced by the cache misses $C_l(m_r + m_w)$ is a significant fraction of the overall traffic $C_d TNN_s$, i.e. $\frac{C_l(m_r+m_w)}{E(m)} \approx 1$, the speedup is high. But once this fraction becomes small, i.e. $\frac{C_l(m_r+m_w)}{E(m)} \ll 1$, the speedup becomes negligible. We have a strong scaling model similar to Amdahl's Law: even many-fold reductions on a small fraction of the overall execution time give only small absolute returns. This explains the bended curves of measured execution time in Figures 5.5 and 5.6 despite the linear decrease in cache misses from Figure 5.8. The performance model formalizes this behavior and fits the bended curves closely to the measured execution times as shown in Figures 5.9 and 5.10

Now that we understand which parameters control the achieved speedup, we can use the

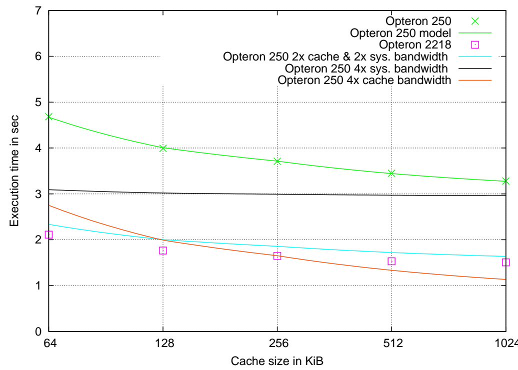



**Figure 5.11.** Performance evaluation of the Opteron 250 against the Opteron 2218 w.r.t the CATS scheme.

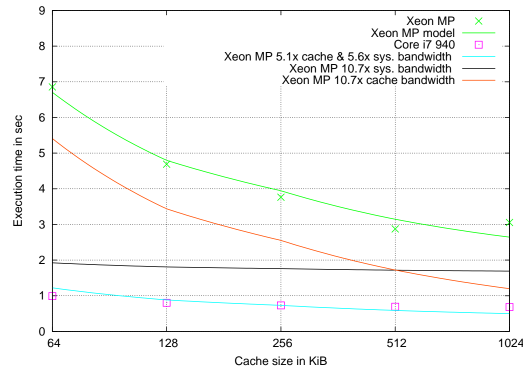



**Figure 5.12.** Performance evaluation of the Xeon MP against the Core i7 940 w.r.t the CATS scheme.

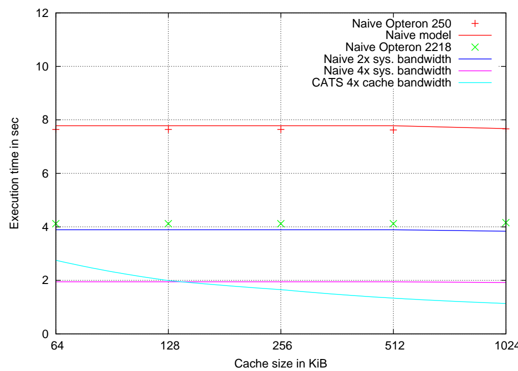



**Figure 5.13.** Performance evaluation of the Opteron 250 against the Opteron 2218: A comparison between the naive and the CATS schemes.

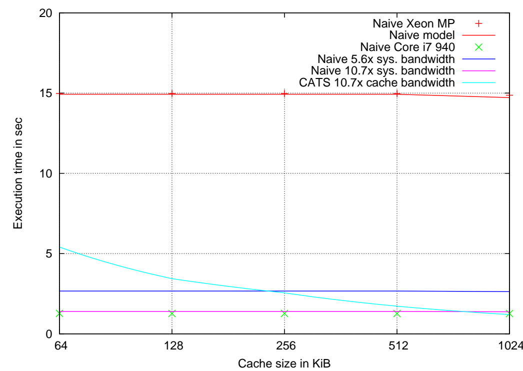



**Figure 5.14.** Performance evaluation of the Xeon MP against the Core i7 940: A comparison between the naive and the CATS schemes.

performance model to estimate the impact of changes in these parameters.

## 5.6 Model Evaluation

With our parameter controlled performance model, we can roughly predict the effects of increasing the system bandwidth or increasing the cache bandwidth on the execution time, and thus evaluate the impact of these system parameters on the performance of stencil computations. First we want to use this feature to validate the model by increasing the parameters of our older test machines, such as to reach the execution times of the newer machines. In a second step, we increase the parameters even further predicting the performance of non-existent hardware.

Figure 5.11 compares the Opteron 250 with the Opteron 2218, while Figure 5.12 looks at the Xeon MP and the Core i7 940. The points in the figure denote the actual execution times while the lines denote the model. From Table 5.1 we compute the ratio between the benchmarked system and cache bandwidths of Opteron 250 and Opteron 2218, we do

the same for the Xeon MP and the Core i7 940, and use these ratios to scale our model parameters. We model three scenarios: scaling $b_{\text{sys}}$, scaling $b_{\text{cache}}$, and scaling both.

If the model is accurate then the scaling with the above ratios of the system and cache bandwidth in the models of the old machines should recover the measured execution times of the new machines. In fact, the Opteron 250 model with doubled cache and system bandwidth comes close to the measured performance of the Opteron 2218 in Figure 5.11 and the Xeon MP model with 5.1x cache and 5.6x system bandwidth scaling comes close to the measured performance of the Core i7 940 in Figure 5.12. This validates our assumption that for stencil computations the cache and system bandwidth parameters matter most and the actual processor architecture is rather irrelevant, even though the Xeon MP design is eight years older and completely different from the Core i7 940 design. Because of this successful validation, we are confident to use this model also for new parameter configurations of non-existent hardware.

On one end of the spectrum, scaling the system bandwidth of the Opteron 250 by a factor of 4 in Figure 5.11 does not benefit the CATS scheme so much. It rather flattens the curve making it similar to the naive performance, because such a great increase in system bandwidth would put it on par with the cache bandwidth. On the other end of the spectrum, a quadrupled cache bandwidth in Opteron 250, accelerates the CATS scheme even beyond the Opteron 2218 performance on large cache sizes. This result is obtained by only changing the cache bandwidth in the CPU, the system bandwidth would still be half that of the Opteron 2218 system. We see that the performance of the CATS scheme reacts very favorably to cache bandwidth scaling even if it is not accompanied by a faster system bus. This is a very cost-efficient way of increasing the overall performance, although it deteriorates the ratio of off-chip to on-chip bandwidth which is usually blamed for bad performance of stencil computations. Instead, we see that performance depends strongly on the implementation of stencil computations, worsening this ratio can actually be a good thing to do.

The advocation of multi-channel memory buses simply comes from the fact that most stencil computations are implemented in a naive way that depends on the system bandwidth for performance. Figure 5.13 shows that doubling or quadrupling the system bandwidth accelerates the naive scheme by almost the same factor. But changing the system bandwidth so radically is a very expensive procedure. In comparison, we see that the inexpensive quadrupled cache bandwidth on CATS still outperforms the enhanced naive scheme by factor 1.8x if the cache size is 1024KiB.

In Figure 5.12, we perform a similar analysis for the old Xeon MP and the new Core i7 940 Intel architectures. The benchmarked system and cache bandwidth ratios between them are 5.6x and 5.1x, respectively (see Table 5.1). We use these factors to scale the Xeon MP

performance model. The predicted CATS performance with 5.6$x$ system and 5.1x cache bandwidth is in fact almost the same as the measured execution times on the Core i7 940. Moreover, we see in Figure 5.12 that further doubling the system bandwidth but leaving the cache bandwidth on the original value would not get us this far. On the other hand, if we leave the very low system bandwidth of the Xeon MP intact, and increase its cache bandwidth to twice that of the Core i7 940, we would still fall short of the Core i7 940 performance but would already beat the much more expensive system bandwidth scaling by the same factor for the 1024KiB cache size.

The situation for the naive scheme on the Xeon MP in Figure 5.14 is very similar to the AMD equivalent from Figure 5.13. The naive schemes benefit proportionally from the system bandwidth scaling; however, for the 1024KiB cache size, the far more inexpensive multiplication of cores in the Xeon MP without changes to the system bandwidth would already deliver superior results.

The discussed relation of system parameters for CATS clearly supports the option of increasing the cache bandwidth rather than the system bandwidth. In current systems cache bandwidth increases automatically with the growing number of cores provided that each core has its own locally connected cache. This scaling option comes with the overhead of keeping a large number of caches coherent; however, CATS features big tiles and requires data synchronization only at their boundaries if they are processed by different threads. Therefore, this synchronization could be performed explicitly with little overhead on a system with non-coherent caches. By further increasing the speed of the local caches, one could quickly obtain enormous speedups in stencil computations using time skewing schemes. One may even reduce the cache size in favor of more cache bandwidth if the discrepancy to the system bandwidth is not too high. If the cache to system bandwidth discrepancy becomes very high, the CATS performance curves become very steep and give bad results for small cache sizes, see Figure 5.12.

Unfortunately, the cost-efficient strategy of deteriorating the ratio of off-chip to on-chip bandwidth through the introduction of faster caches does not help the naive codes. So we are in a dilemma here. Using clever schemes, we can increase the performance of stencil computations radically by the simple scaling of the aggregate cache bandwidth, but all naive codes would suffer in this situation and even more severely demand an increase in system bandwidth. Therefore, concerning iterative stencil computations, the bandwidth wall problem is only partially a hardware issue, more importantly we have a software issue of ineffective implementations in many codes. The expensive scaling of the system bandwidth through multi-channel memory interfaces could stop without deteriorating performance if more codes would change the way iterations of stencil computations are implemented. Of course, not all applications can benefit from time skewing, so one would need to know the fraction of iterative stencil computations in the application mix to

determine which amount of system and cache bandwidth would give the best performance per cost ratio on average.

## 5.7 Conclusion

We have examined the impact of system and cache bandwidth on the naive and the cache accurate time skewing (CATS) scheme for iterative stencil computations. The schemes exhibit almost completely opposite behavior. While the naive scheme requires high system bandwidth for performance, the same stencil computation can be performed with a time skewing scheme much faster if only the cache bandwidth in the CPU is increased. The latter option gives by far the more cost-efficient performance gains, e.g. we could execute on the ten years old Xeon MP as fast as on a Core i7 940 if only sufficient cache bandwidth in the Xeon MP were provided without the need for any improvement of its outdated system bus. So the paradoxical conclusion is that for iterative stencil computations further deteriorating the ratio of off-chip to on-chip bandwidth is the cheapest way to higher performance. Unfortunately, the situation is more complex in practice because not all stencil computations occur in iterations and many of them operate with varying rather than constant coefficients which puts additional strain on the system bus. In future, we want to extend the performance model so that it allows to predict the behavior for more computational patterns. However, even the restricted model makes it clear that a solution to the bandwidth wall problem should not be sought solely in system bandwidth scaling, because it is not necessarily the limiting factor even if the data is much bigger than the caches and has to be accessed many times.

# Chapter 6

# Cache Oblivious Parallelograms in Iterative Stencil Computations

In this Chapter, we present a novel cache oblivious scheme for iterative stencil computations on symmetric multiprocessing (SMP) memory systems, called *CORALS*. Despite the tremendous cache miss rate reduction by cache oblivious stencil algorithms, previous realizations of these approaches have shown a slight improvement against the naive scheme whereas CORALS achieves a remarkable speedup without fine tuning for the specific characteristics of each architecture. In particular, 2D CORALS achieves about 10x speedup over an optimized naive scheme on a domain of 128 million double precision elements running on a quad-core Xeon X5482 machine. The performance amounts to 47% of the measured computational machine peak. CORALS also clearly outperforms more general transformation tools.

## 6.1 Previous Work

Frigo and Strumpen [17] introduced a cache oblivious stencil scheme that divides the iteration space recursively into smaller and smaller space-time tiles and thus generates high temporal locality on all cache levels without knowing their sizes. The cache misses are greatly reduced leading to the desired reduction of system bandwidth requirements, however, the performance gains are relatively small in comparison to this reduction. Strumpen and Frigo [56] report a 2.2x speedup against the naive implementation of a 1D Lax-Wendroff kernel on a IBM Power5 system for periodic and constant boundary conditions after optimizing the software aspects of the scheme. After multifold optimizations and parameter tuning Kamil et al. [32] achieve a 4.17x speedup on the Power5 (15 GB/s theoretical peak bandwidth), 1.62x on an Itanium2 (6.4 GB/s) and 1.59x on an Opteron (5.2

GB/s) system for a 7-point stencil (two distinct coefficient values) on a $256^3$ domain for periodic boundary conditions. However, for constant boundary conditions the optimized cache oblivious scheme is only faster on the Opteron achieving a 2x speedup at best. The compared naive code is optimized with ghost cells and compiled with optimization flags.

The above optimizations of the cache oblivious scheme are all directed at single-threaded execution. Frigo and Strumpen later analyzed multi-threaded cache oblivious algorithms [18]. One example deals with the cache misses of a 1D stencil code with parallel tile cuts. Blelloch et al. [6] discuss the construction of nested parallel algorithms with low cache complexity in the cache oblivious model for various algorithms including sparse matrix vector multiplication. However, these are mainly theoretical papers and we do not know of any parallel, high performance cache oblivious implementations of stencil computations based on these ideas.

The next section (Section 6.2) presents our new algorithm in detail. We start with the description of our cache oblivious approach in multiple sub-sections. Section 6.3 discusses the results in double precision on 2D and 3D domains. Constant stencils, banded matrices and an FDTD solver are presented.

## 6.2 Cache Oblivious Parallelograms in Iterative Stencil Computations

The scheme starts by covering the entire space-time with a single large tile to which we assign all the available threads (Section 6.2.1). Then we run some preprocessing that generates data for the load-balancer (Section 6.2.4). The initial tile is divided recursively into a high number of identical base tiles for which we stop the recursion. During the recursion, the division tries to distribute the threads and thus assign fewer and fewer threads to the sub-tiles (Section 6.2.2). The thread distribution is governed by the load balancer (Section 6.2.4). On each base tile, the kernel containing the actual stencil computation is invoked with a single thread even if more threads are still assigned to the base tile. So all parallelization must occur through the thread distribution during the recursive division, the kernel execution itself is single-threaded. In higher dimensional space-time this task is easier (Section 6.2.3). The choice of the base tile and other internal parameters are discussed in Section 6.2.5.

### 6.2.1 Parallelograms in 2D

As discussed in Section 6.1, the attempts to extract high absolute performance from the original cache oblivious stencil scheme were not very successful, although cache misses

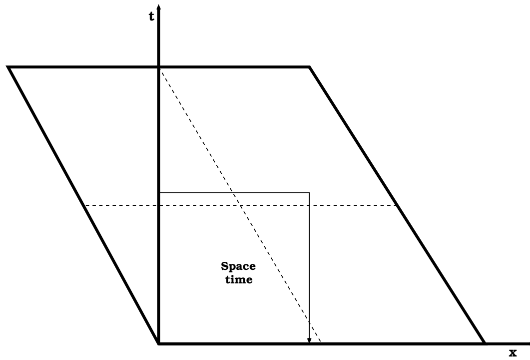



**Figure 6.1.** The entire space-time covered by a large initial parallelogram. The work-load on the sub-parallelograms is substantially different.

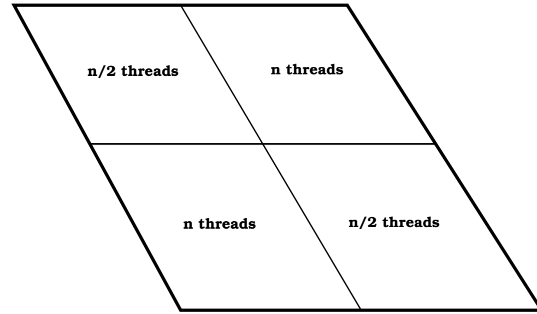



**Figure 6.2.** The thread distribution for a 2D cut of an interior parallelogram. The upper left and lower right sub-parallelograms are independent and run in parallel.

are significantly reduced and hardware specific optimizations were applied. We think that one of the problems lies in the irregularity of the generated space-time tiles, because the compiler and hardware generally perform much better on regular data structures. Therefore, a main design aspect of CORALS has been the preservation of the theoretic asymptotic behavior of the original cache oblivious stencil scheme while utilizing only regular execution patterns: CORALS applies the hierarchical decomposition idea to a single-form space-time tile, namely a parallelogram. The following parallelization and data locality strategies are a consequence of this decision.

Parallelograms have a favorable surface area to volume ratio. They have the advantage that we can iterate over the interior points with simple nested loops where each loop always executes the same number of runs, only the bounds are skewed with respect to the time. This allows an efficient vectorized execution with explicit control of data alignment. As we have only a single tile form, only one such specialized kernel must be implemented. Moreover, parallelograms split easily into identical sub-parallelograms and also allow a regular parallelization.

As there are only parallelograms in the scheme we start with a large parallelogram that covers the entire space-time, see Figure 6.1. This figure shows that an exterior coverage of the domain leads to different work-loads inside the tiles. Section 6.2.4 on load-balancing discusses this in detail. In the following we assume that we deal with parallelograms that lie completely inside the space-time.

Figure 6.2 shows the canonical subdivision scheme for the parallelogram. The dimensions of the initial parallelogram are made divisible by a large power of two, such that we can perform correspondingly many subdivisions without having to half an odd number. Consequently, the sub-parallelograms are identical in shape. The figure also shows the thread distribution for an interior parallelogram. The upper left and lower right sub-

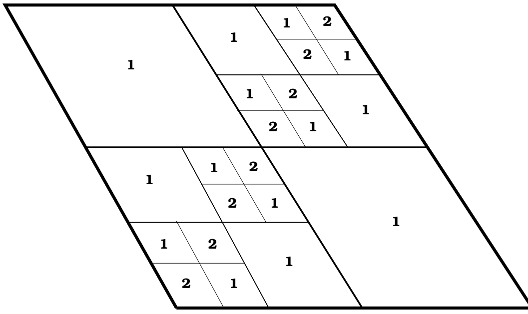


**Figure 6.3.** Recursive application of the 2D subdivision from Figure 6.2 in case of two threads. Sub-parallelogram assigned with one thread execute in parallel with another sub-parallelogram.

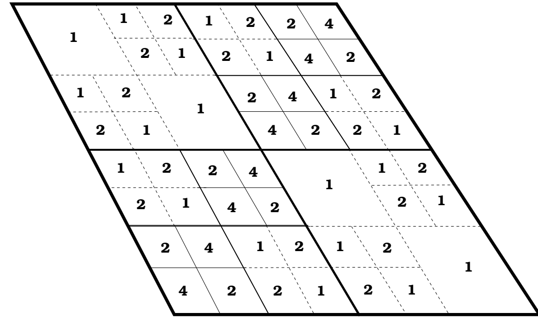


**Figure 6.4.** Recursive application of the 2D subdivision from Figure 6.2 in case of four threads. Dotted lines show the additional sub-divisions in comparison to two threads from Figure 6.3.

parallelograms are independent of each other, so given $n$ threads, each of them is assigned $n/2$ threads and they are executed in parallel.

Frigo and Strumpen suggest a spatial trapezoid parallelization in their multi-threaded cache oblivious 1D stencil algorithm [18] and this is also the usual choice in cache aware time skewing schemes, but 2D parallelograms are not suitable for that. The spatio-temporal parallelization is a simple solution to this problem.

### 6.2.2 Parallelism and Locality

In Figure 6.2 two of the four sub-parallelograms are executed in parallel, so in case of $n = 2$ threads the overall execution time would be reduced to $\frac{3}{4}$ rather than $\frac{1}{2}$. However, the cache oblivious scheme performs a recursive subdivision of the tiles, so wherever we have more than one thread per sub-parallelogram, it is further divided in the same fashion. Figure 6.3 shows the thread distribution after three division steps in case of $n = 2$ threads. We see that almost the entire domain is parallelized and only the small blocks on the diagonal still require further division for parallelization. After a few division steps, the reduction of the overall execution time converges quickly to $\frac{1}{2}$ according to a geometric series.

Let us formalize the above reasoning for two threads at first considering the effects of parallelism only. All parallelization must be made explicit through subdivision, so the processing of an undivided parallelogram takes the same time no matter how many threads are assigned to it. Let the initial undivided parallelogram have base width $w$ and height $h$ then its execution time is $wh$ in an appropriate time unit. After the first division, two of the sub-parallelograms run in parallel so the overall time is $\frac{3}{4}wh = \frac{1}{2}(wh/2 + wh)$, where $\frac{1}{2}(wh/2)$ corresponds to the sub-parallelograms with one thread assigned and $\frac{1}{2}wh$ corresponds to the two sub-parallelograms with two threads assigned, see Figure 6.2. In

the next step each of the two sub-parallelograms with two threads assigned undergoes the same parallelization as the initial parallelogram so the $\frac{1}{2}wh$ is replaced by $\frac{3}{4}(\frac{1}{2}wh)$ or equivalently $wh$ is replaced by $\frac{3}{4}wh$ as before, giving $\frac{1}{2}(wh/2 + \frac{1}{2}(wh/2 + wh))$ overall. The following division replaces the last $wh$ in the same fashion and we obtain a recursive formula for the execution time in dependence on the division depth:

$$\begin{aligned}
\text{execT}(0) &= wh \\
\text{execT}(1) &= \frac{1}{2}(wh/2 + wh) \\
\text{execT}(2) &= \frac{1}{2}(wh/2 + \frac{1}{2}(wh/2 + wh)) \\
\text{execT}(a) &= \frac{1}{2}(wh/2 + \frac{1}{2}(wh/2 + \frac{1}{2}(\ldots))) \\
&= wh/2\left(\frac{1}{2} + \ldots + (\frac{1}{2})^a + 2(\frac{1}{2})^a\right) \\
&= wh/2\left(1 + (\frac{1}{2})^a\right) . \qquad (6.1)
\end{aligned}$$

For two threads, the geometric series converges quickly to the ideal execution time reduction by $\frac{1}{2}$.

In cache aware time skewing schemes, flat parallelization strategies are applied [72, 35, 44]. The cache sizes are known, so it is clear when it is better to parallelize the execution of the sub-tiles, forcing them into different caches, and when to leave them in the same cache for better data locality and process them sequentially with a single thread. In the cache oblivious case we do not have this information so on the one hand we must ensure that the parallelism really speeds up the computation, as demonstrated above, and on the other hand we must maximize the tile sizes that are processed by a single thread for best data locality within the same cache (we assume the scalable scenario where caches are not shared between cores). A similar reasoning as above shows that the second condition is also fulfilled. The first division assigns already half of the domain to the local execution by a single thread, the next division adds a half of the remaining half leading to a geometric series again $\frac{1}{2} + \frac{1}{4} + \ldots$. At some stage the tile bases are smaller than the cache so the parallelization will force its sub-tiles into different caches destroying the data locality *between* the sub-tiles, but this happens only in a small part of the domain that correspond to the trailing end of the above series. In conclusion, the scheme preserves as much data locality as possible while converging to the full parallel speedup according to a geometric series.

The parallelization in case of more threads is not much different. Figure 6.4 shows how the division simply continues in all parts of the domain as long as more than one thread is assigned to a parallelogram. If we stop the recursion at a certain level then we are left with

one diagonal of small parallelograms where still four threads are assigned and multiple thin sub-diagonals of parallelograms where still two threads are assigned. This only adds more trailing factors in addition to the already existing $(\frac{1}{2})^a$ in formula (6.1), so the properties of the geometric convergence to the full speedup of 4x in case of four threads remains unchanged. The geometric convergence property also holds for an arbitrary number of threads but we can not expect perfect strong scaling with the thread count because the more threads there are, the more divisions are necessary to arrive at the local single thread execution of a tile. However, for weak scaling the domain would also grow, increasing the number of divisions before we reach a fixed base parallelogram size, just as required above.

All parallelograms in Figures 6.3 and 6.4 that have been assigned a single thread are also further divided for the cache oblivious data locality but no more parallelism needs to be extracted in these divisions. These divisions are not included in the figures to facilitate the reasoning about the generated parallelism. Ultimately the entire initial parallelogram is divided into a large number of identical base parallelogram on which we stop the recursion and call the kernel with the stencil computation.

### 6.2.3 Parallelograms in higher dimensions ($m$D)

This section explains our scheme for an arbitrary dimension $m$ of the space-time. We explain the differences to the 2D iteration space and refer for analogy to the previous 2D figures.

In an $m$-dimensional space-time, we have $m - 1$ spatial dimensions formed by a tensor product of the individual spatial dimensions. The space-time tiles in 2D are parallelograms, in 3D parallelepipeds and in general $m$-parallelotopes in mD. The projection of an $m$-parallelotope onto the time axis and one of the spatial axis always gives a parallelogram as depicted in Figure 6.1. So all the spatial dimensions are skewed with respect to time and in analogy to 2D we can create an $m$-parallelotope large enough to cover the entire space-time, see Figure 6.1.

For the properties of the recursive division and the parallelization, the skewing and the absolute sizes of the different tile dimensions play no role, so instead of an $m$-parallelotope one can also think of a simple $m$-hypercube in the following, where all cuts are axis aligned. Figure 6.2 shows a 2D division into $2^2 = 4$ sub-parallograms. With a single cut we could also make a 1D division in 2D delivering $2^1 = 2$ identical sub-parallelograms. The $m$-parallelotope allows kD cuts with $k = 1, \ldots, m$. A kD cut of the $m$-parallelotope gives $2^k$ identical sub-parallelotopes. The number of created sub-parallelotopes and their following parallelization does not depend on the space-time dimension $m$ but on the cut dimension $k$. The reason for considering cuts of different dimensions is that depending on

the parameter settings we want to stop cutting one dimension at a certain tile size, e.g. the unit stride dimension, while other dimension should still be cut. This leads to the applications of different cuts during the recursive division of tiles.

The parallelization of the 2D cut requires $2+1=3$ execution stages (with synchronization in between) with the following number of independent sub-parallelograms in each stage: $\binom{2}{0}=1, \binom{2}{1}=2, \binom{2}{2}=1$, cf. Figure 6.2. Similarly, for a kD cut we have $k+1$ stages where the series of independent sub-parallelotopes in each stage is: $\binom{k}{j}, j=0,\ldots,k$. So with higher dimensional cuts, it is much easier to extract parallelism from the division scheme, e.g. a 4D cut (applicable to 3D spatial domains and higher) gives a series of 1, 4, 6, 4, 1, i.e. after finishing the first sub-parallelotope there are already four independent sub-parallelotopes that can be executed in parallel.

In $m$D space-time, it is also possible to extract purely spatially independent sub-tiles. The independence of the upper left and the lower right parallelogram in Figure 6.2 is spatio-temporal. But if the time dimension is very small, e.g. only $T=10$ iterations of the stencil computation are required, then we do not want to cut it further, and a spatial 1D cut would only generate two dependent tiles with no opportunity for parallelism. However, in mD space-time with $m>2$, simultaneously cutting multiple spatial dimensions produces spatially independent sub-tiles even if the time dimension is uncut.

The better parallelization potential of higher dimensional cuts means that in the recursive division, we can more quickly distribute threads and as such less depth is needed to reach the local single thread execution on a tile. Figures 6.3 and 6.4 depict the recursive division with 2D cuts. With higher dimensional cuts, the size of the tiles that still need further division decreases faster and thus the geometric convergence (formula (6.1)) to the full speedup is also faster.

### 6.2.4 Load-Balancer

The thread distribution from Figure 6.2 assumes that the parallelized execution of the upper left and the lower right sub-parallelogram have the same work-load, so assigned with the same number of threads, they will finish at approximately the same time without creating idle time at the following synchronization point. Because of our exterior structure (Figure 6.1) many parallelograms do not have the same work-loads and this results in some idle time at the synchronization points. The CORALS scheme is fast even with this handicap. This section describes the further performance enhancement of the load-balancer.

The load-balancer distributes the threads to the parallelized sub-parallelogram according to the actual work-load. To determine the work-load, we execute a preprocessing step

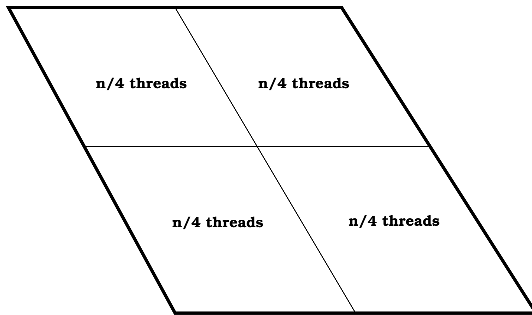



**Figure 6.5.** The thread distribution for a 2D cut during the preprocessing, see Algorithm. 6. No sub-parallelogram dependencies need to be respected here.

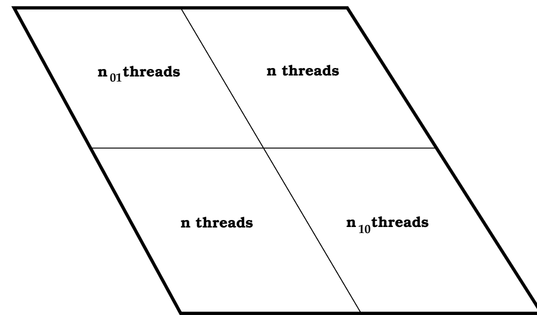



**Figure 6.6.** The thread distribution for a 2D cut during the stencil computation with the load-balancer, see Algorithm 7. Usually we have $n_{01} + n_{10} = n$ and the corresponding two sub-parallelograms execute in parallel.

**Algorithm 6 The CORALS preprocessing function takes a recursive descent through the sub-tiles. The preprocessing evenly distributes the current thread range onto the sub-tiles without explicit synchronization (Figure 6.5).**

```
int CORALSpreprocess(tile)
{
  create divSet from tile: the set of divisible dimensions;
  if( divSet is empty ) {
    pointsSum= countInteriorPointsOn(tile);
  } else {
    based on divSet divide tile into a subTileSet;
    distribute tile.threadRange evenly onto sub-tiles;
    pointsSum= 0;
    forall( subTile ∈ subTileSet ) {
      if( tid ∈ subTile.threadRange ) {
        pointsSum+= CORALSpreprocess(subTile);
      }
    }
  }
  if( tile.threadRange == {tid} ) {
    assign pointsSum to global tile pointsSum;
  } else {
    atomicAdd of pointsSum to global tile pointsSum;
  }
  return pointsSum;
}
```

that determines the number of interior points for the entire parallelogram hierarchy. This preprocessing uses the same recursive division only with a simpler thread distribution scheme (Figure 6.5), because no sub-tile dependencies have to be observed here. The interior points are counted on the base tiles and summed up recursively on the way up, see Algorithm 6.

During the actual stencil computation (Algorithm 7), the numbers of the interior points of the independent sub-parallelograms are put in relation to their overall sum and the available threads are distributed according to these ratios. Currently, we use a simple distribution model balancing pairs of independent sub-parallelograms. It has the advantage that the same model can be applied for cuts of any dimension. In Figure 6.6, this means that normally $n_{01} + n_{10} = n$ and $n_{01}, n_{10}$ are chosen such that the ratios $n_{01}/n$, $n_{10}/n$ approximate the corresponding ratios of the numbers of the interior points to their sum. If one of these ratios is very small, e.g. one of the sub-parallelograms contains only a few interior points, then the choice $n_{01} := n$ and $n_{10} := n$ is better. It postpones the parallelization to the next division level in favor of reducing the idle time at the synchronization point. We make this choice if more than 20% of the available work capacity would be wasted on waiting at the synchronization point.

**Algorithm 7 The CORALS stencil computation. Both preprocessing (Algorithm 6) and stencil functions take the same recursive descent through the sub-tiles, only the parallelization is different. The preprocessing evenly distributes threads onto the sub-tiles with no explicit synchronization, whereas the stencil computation distributes the threads according to the precomputed number of interior points in each sub-tile and respects sub-tile dependencies with explicit synchronization of the current thread range (Figure 6.6).**

```
CORALScompute(tile)
{
  create divSet from tile: the set of divisible
  dimensions;
  if( divSet is empty ) {
    executeStencilComputationOn(tile);
  } else {
    based on divSet divide tile into a subTileSet;
    load-balance tile.threadRange on sub-tiles;
    based on divSet create syncTileSet;
    forall( subTile ∈ subTileSet ) {
      if( tid ∈ subTile.threadRange ) {
        CORALScompute(subTile);
      }
      if( subTile ∈ syncTileSet ) {
        synchronize threads from tile.threadRange;
      }
    }
  }
}
```

### 6.2.5 Internal Parameters

Our scheme has several internal parameters that are not exposed to the user and their general setting is explained in this section. By tuning these parameters we achieve higher performance, but since the optimized naive scheme and PluTo run with automatic parameter settings, tuning our parameters would be unfair. Instead we use fixed values in all evaluations.

From the CORALS description (Section 6.2), we already know that the recursion continues until we reach a certain base tile size. In theory, we could continue the recursion down to individual space-time points but practically this is a bad idea, as more work would be spent on the control logic than the actual computation. So we choose a default base size of 8 for all dimensions other than x and t. The x-dimension size is set to a larger value because it is the unit stride dimension where spatial data locality matters most, and the t-dimension size is set to a larger value because it controls directly the temporal data locality within the base tile. Both values are inherited from the multi-threaded base size which we explain next.

The multi-threaded base size determines in the recursive division when to stop the parallelization of sub-tiles even though multiple threads are still assigned to the parent tile. In Figure 6.3 we see that, in principle, the recursive parallelization on the diagonal tiles can continue infinitely. It definitely stops at the tile base size described above but it makes sense to stop the parallelization even earlier. Here the reason is not the overhead of control logic but the disproportionate costs of exchanging data between the deepest memory level (L1 cache in current architectures) of two distinct cores in comparison to the available bandwidth on this level. In other words, once a tile fits into the deepest memory level, a single-threaded execution is faster than the parallel execution on sub-tiles plus the collection of the results in one core, which is necessary for further processing.

We pick a heuristic memory size value $M_{\text{stop}}$ and compute the *spatial* multi-threaded base size dimensions such that the corresponding data would fit therein with the x dimension being a factor $X_{\text{strech}}$ larger than the others. The multi-threaded t-base size is set equal to the multi-threaded x-base size and we have already explained why these two are assigned larger values. In 3D space-time, we have $M_{\text{stop}}$:= 32KiB, $X_{\text{strech}}$:= 2 and in 4D $M_{\text{stop}}$:= 128KiB, $X_{\text{strech}}$:= 10. Not surprisingly in 4D we want to stop the parallelization earlier because in case of a parallel execution, there are more sub-tiles that require an expensive collection process from the deepest memory level of multiple cores.

**Table 6.1.** Hardware configurations of our test machines. The machines have been chosen such that one, the Opteron, has a modest ratio between measured system and cache bandwidth, while the other, the Xeon, has a high ratio. This ratio is the main source of acceleration of time skewing against naive schemes.
The measured bandwidth numbers have been obtained with the RAMspeed benchmarking tool with 4 threads and SSE reads. The measured double precision (DP) FLOPS numbers come from our own SSE benchmarks. For the peak DP number we perform independent multiply-add operations on registers, for the stencil DP number we run the inner stencil computation (products and sums) on registers. This value is lower because of the read-after-write dependencies in the computation.

| Brand | AMD | Intel |
|---|---|---|
| Processor | Opteron 2218 | Xeon X5482 |
| Code-named | Santa Rosa | Harpertown |
| Frequency | 2.6 GHz | 3.2 GHz |
| Number of sockets | 2 | 1 |
| Cores per socket | 2 | 4 |
| L1 Cache per core | 64 KiB | 32 KiB |
| L2 Cache per core | 1 MiB | 3 MiB |
| Operating system | Linux 64 bit | Linux 64 bit |
| Parallelization | 4 pthreads | 4 pthreads |
| Vectorization | SSE2 | SSE2 |
| Compiler | g++ 4.3.2 | icpc 11.1 |
| Measured L1 Bandwidth | 79.3 GB/s | 194.6 GB/s |
| Measured L2 Bandwidth | 40.6 GB/s | 64.2 GB/s |
| Measured Sys. Bandwidth | 11.2 GB/s | 6.20 GB/s |
| Measured Peak DP FLOPS | 20.8 G | 40.8 G |
| L2 Band./Sys. Bandwidth | 3.6 | 10.4 |
| Peak DP/(Sys. Band./8B)<br>Balanced arith. intensity for Sys. | 14.9 | 52.6 |

## 6.3 Results

We compare the results of the following three schemes for iterative stencil computations:

- **NaiveSSE:** Our own parallelized (pthreads) and vectorized (SSE2) naive stencil scheme as described in Section 4.3.1.
- **PluTo [7]:** Code transformed by the automatic parallelizer and locality optimizer for multicores; PluTo, version 0.4.2. We use the original code examples and modify them from constant to variable stencil where necessary.
- **PeakDP:** The measured computational peak in double precision. We obtain this value by performing a sequence of independent multiply-add operations in registers. PeakDP models the absolute upper bound for any computation on a machine. The ultimate goal of optimized stencil computations is to achieve a high fraction

of this peak as no optimization of stencil codes will reach this value because of the dependency between the stencil operations.

- **CORALS:** Our cache oblivious parallelograms scheme with the internal parameters described in Section 6.2.5 and pthreads parallelization. The innermost loop of the kernel is vectorized (SSE2). Preprocessing time is included.

For all 2D and 3D domains, the codes are recompiled with compile-time known domain sizes. For CORALS, this is rather irrelevant but the naive scheme and PluTo benefit from this procedure. All methods use four threads.

Test applications comprise constant and variable stencils in 2D and 3D with 0.5 to 128 million double precision elements. In 2D, we have squares ranging from $706^2$ to $11282^2$ elements and in 3D, cubes from $80^3$ to $500^3$. In case of constant stencils, this amounts to a memory consumption of up to 2GiB for the two vectors, and in case of variable stencils we use at most 32 million elements consuming 0.5GiB plus 1.75GiB for the matrix in 3D. We use a 5-point stencil in 2D and a 7-point in 3D. The number of iterations is either $T = 100$ (solid graphs in the figures), or $T = 10$ (dashed graphs in the figures). The last stencil application is the FDTD 2D example that comes with PluTo.

All figures show the execution time in seconds against the number of elements in millions with both axes being logarithmic. The number of elements doubles between two consecutive graph points, but the doubling is not exact because of the root operations involved in computing a square or cube with a predefined number of elements.

The hardware configuration of our two test machines is listed in Table 6.1. For pluto-0.4.2, we experimented with different options and finally used -tile -l2tile -multipipe -parallel -unroll -nonuse although the last two options do not make a difference in performance in our examples. For the 3D examples, we eventually dropped -l2tile as the transformation process was taking hours without gaining any performance in the end. The PluTo transformed code is compiled with the additional option -fopenmp to enable OpenMP support. We try to get the most out of the PluTo code by recompiling with compile-time known domain sizes and the aggressive icpc compiler settings from Table 6.1 which requires about 15 minutes compilation time for every domain size.

### 6.3.1 Constant Stencil

Figure 6.7 shows the execution times on the Opteron 2218 for 2D spatial domains. It is difficult to beat an optimized naive code in this setting because the balanced stencil intensity from system memory is just 8.2 on this machine, see Table 6.1. This means it suffices to have 8.2 double operations in the kernel for every double read from system

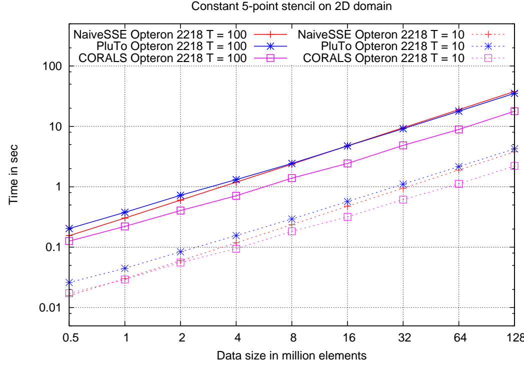



**Figure 6.7.** Timings of the Opteron 2218 with constant stencils in 2D. GFLOPS for 128 million elements with $T = 100$: NaiveSSE Opteron 3.4, PluTo Opteron 3.6, CORALS Opteron 6.5 (31% of PeakDP).

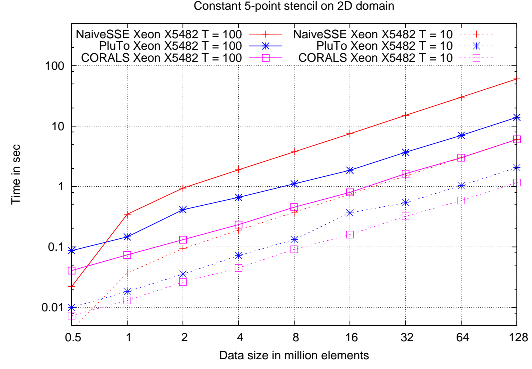



**Figure 6.8.** Timings of the Xeon X5482 with constant stencils in 2D. GFLOPS for 128 million elements with $T = 100$: NaiveSSE Xeon 1.9, PluTo Xeon 8.2, CORALS Xeon 19.1 (47% of PeakDP).

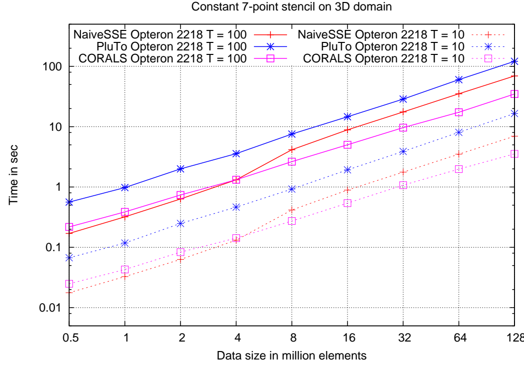



**Figure 6.9.** Timings of the Opteron 2218 with constant stencils in 3D. GFLOPS for 128 million elements with $T = 100$: NaiveSSE Opteron 2.4, PluTo Opteron 1.5, CORALS Opteron 4.8 (23% of PeakDP).

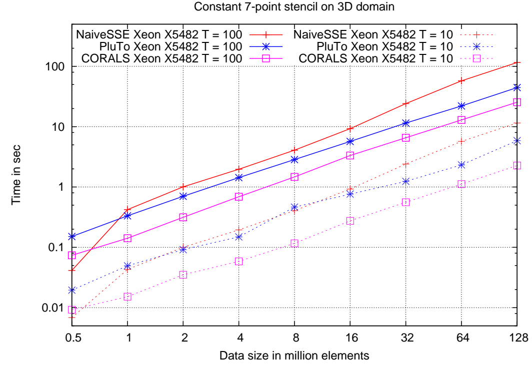



**Figure 6.10.** Timings of the Xeon X5482 with constant stencils in 3D. GFLOPS for 128 million elements with $T = 100$: NaiveSSE Xeon 1.4, PluTo Xeon 3.7, CORALS Xeon 6.5 (16% of PeakDP).

memory to avoid memory stalls. Because our kernel also needs to write out a value with every stencil computation, the stencil intensity 8.2 doubles to 16.4. The 5-point constant 2D stencil has 9 double operations, and if the cache can hold four lines (3 input plus 1 output) of the 2D domain simultaneously, then 4 values come from the cache and only one comes from the system memory on average. So the kernel is memory-bound by only a small factor $16.4/9 \approx 1.82$. Even for this small factor, CORALS shows superior results in Figure 6.7 and the advantage grows with larger domain sizes. PluTo on the other hand becomes barely better than the naive scheme for large domain sizes and $T = 100$ iterations and loses the comparison for $T = 10$ iterations.

In case of the 3D spatial domain on the Opteron (Figure 6.9), the naive scheme becomes unbeatable when four slices of the domain fit into the cache, because the 7-point stencil computation requires 13 double operations, so accounting for both reading and writing we get: $16.4/13 \approx 1.26$, i.e. the computation and bandwidth requirements are almost

balanced in the naive scheme and the performance difference to CORALS reveals its small control logic overhead. As soon as the four slices of the domain do not fit into the cache of 1MiB, which occurs first for the 8million $= 200^3$ elements domain $(4 \cdot 200^2 \cdot 8\text{B} = 1.28\text{MB} > 1\text{MiB})$, CORALS wins easily against the naive scheme again. PluTo does not perform good in 3D on the Opteron.

The 2D situation on the Xeon (Figure 6.8) is very different from the Opteron. First, we see that the naive scheme shows an excellent performance for 0.5 million elements. In this case two full vectors consume $0.5\text{million} \cdot 2 \cdot 8\text{B} = 8\text{MB}$ that fit completely into the 12MiB L2 cache of the Xeon, so all processing happens in cache. For all bigger domain sizes, this is not the case and hence the naive scheme becomes slow again.

PluTo shows much better performance on the Xeon than the Opteron. However, CORALS is still better by more than a factor of 2. On large domains, it completes 100 iterations in approximately the same time as the naive scheme needs for 10 iterations. The computational performance of 19.1 GFLOPS on the 128 million elements domain reaches 47% of the peak machine performance of 40.8 GFLOPS on this kernel (cf. Table 6.1). While the synthetic benchmark operates only on registers with no memory access, CORALS alternates between two 1GiB large vectors in this test. This is an excellent performance result for a highly memory-bound multi-dimensional kernel and demonstrates the real potential of time skewing schemes.

The 3D results on the Xeon (Figure 6.10) show that it is more difficult to extract data locality in 3D. PluTo beats the naive scheme by a much smaller factor than in 2D, although the situation improves for larger domain sizes. CORALS still shows a significant advantage over PluTo, but the absolute performance is 6.5 GFLOPS which is clearly lower than in 2D. Finally, as expected from the above discussion, the fast execution of the naive scheme on the 0.5 million elements domain is also present here.

In summary, we observe that PluTo performs well on the Xeon where the kernel is highly memory-bound and the icpc compiler is used, on the Opteron where the kernel is only slightly memory bound, it loses the comparison against the naive scheme. CORALS delivers clearly superior results overall, only where the machine characteristics favor the naive scheme, CORALS becomes slightly inferior.

### 6.3.2 Banded Matrix

The situation on the Opteron for the banded matrix is similar to the constant stencil. In 2D (Figure 6.11), PluTo loses to the naive scheme by a small margin, while CORALS maintains a consistent advantage that, however, is significantly larger in this banded matrix case, cf. Figure 6.7. In 3D on the Opteron (Figure 6.13), PluTo is much slower than

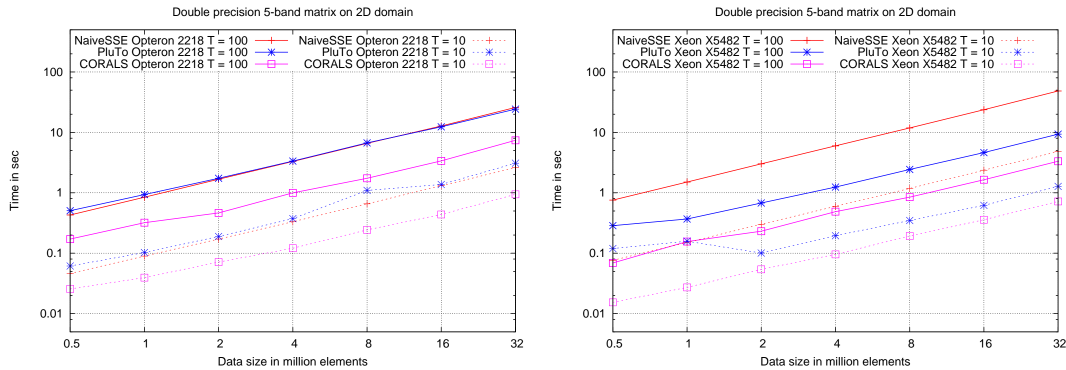



**Figure 6.11.** Timings of the Opteron 2218 with a banded matrix in 2D. GFLOPS for 32 million elements with $T = 100$: NaiveSSE Opteron 1.1, PluTo Opteron 1.2, CORALS Opteron 3.9 (19% of PeakDP).

**Figure 6.12.** Timings of the Xeon X5482 with a banded matrix in 2D. GFLOPS for 32 million elements with $T = 100$: NaiveSSE Xeon 0.6, PluTo Xeon 3.1, CORALS Xeon 8.7 (21% of PeakDP).

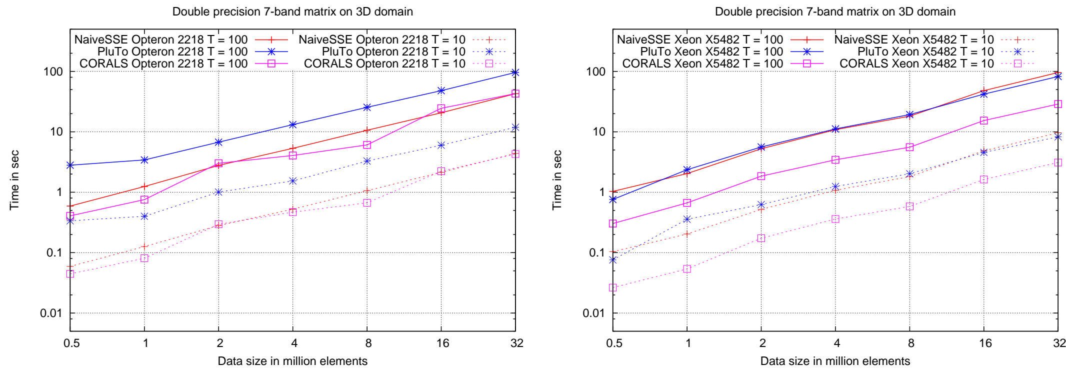



**Figure 6.13.** Timings of the Opteron 2218 with a banded matrix in 3D. GFLOPS for 32 million elements with $T = 100$: NaiveSSE Opteron 1.0, PluTo Opteron 0.4, CORALS Opteron 1.0 (5% of PeakDP).

**Figure 6.14.** Timings of the Xeon X5482 with a banded matrix in 3D. GFLOPS for 32 million elements with $T = 100$: NaiveSSE Xeon 0.4, PluTo Xeon 0.5, CORALS Xeon 1.4 (3% of PeakDP).

the naive scheme, while CORALS performs on average slightly better. This is the only figure where CORALS shows some considerable irregularity without a consistent speedup against the naive scheme.

On the Xeon in 2D (Figure 6.12), PluTo outperforms the naive scheme again by a large margin, while CORALS further improves on that. The advantage for 100 iterations is much higher than for 10 iterations, it even suffices to significantly surpass 10 iterations of the naive scheme. In 3D (Figure 6.14), the superiority of CORALS is equally high for 10 and 100 iterations, while PluTo and the naive scheme perform similarly.

In summary, PluTo gives good results on the Xeon in 2D again, but otherwise it is worse than the naive scheme, in particular, on the Opteron in 3D. CORALS dominates in all cases except on the Opteron in 3D where results are still better than naive and PluTo.

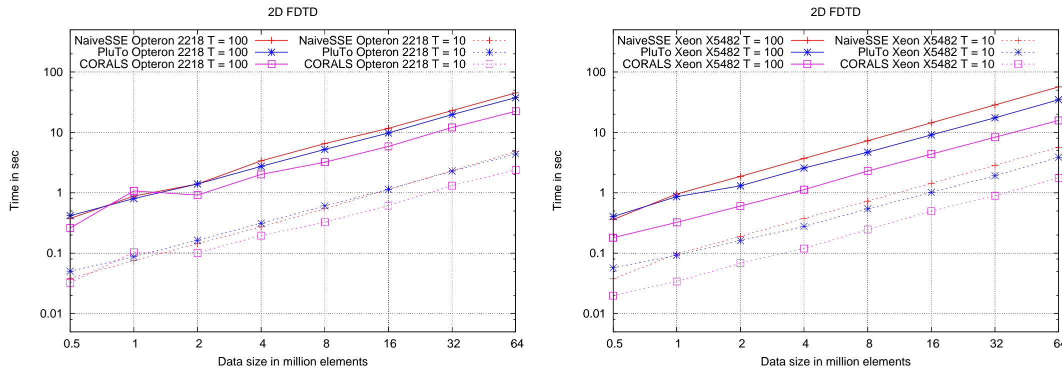



**Figure 6.15.** Timings of the Opteron 2218 for FDTD in 2D. GFLOPS for 64 million elements with $T = 100$: NaiveSSE Opteron 1.6, PluTo Opteron 1.9, CORALS Opteron 3.1 .

**Figure 6.16.** Timings of the Xeon X5482 for FDTD in 2D. GFLOPS for 64 million elements with $T = 100$: NaiveSSE Xeon 1.2, PluTo Xeon 2.0, CORALS Xeon 4.4 .

### 6.3.3 Application: FDTD Solver

The previous sections analyzed basic stencil computations on a scalar domain with constant or variable weights in detail. Here we look at a variation of these basic computations, namely a vector valued problem with in-place updates. The 2D Finite Difference Time Domain (FDTD) electromagnetic solver [61] is often used to demonstrate the efficiency of time skewing schemes (see Section 4.4.6). We use the sample code from PluTo [7]. PluTo can fuse and vectorize the loops automatically while CORALS and the naive scheme require us to explicitly fuse them and vectorize the unit stride loop manually.

Figure 6.15 and 6.16 show the results for Opteron and Xeon, respectively. On the Opteron, the results are comparable to a slower version of the constant stencil in 2D (Figure 6.7), with PluTo and the naive scheme performing similarly. PluTo is a bit faster for 100 iterations and the naive scheme is a bit faster for 10 iterations on average. CORALS shows a mediocre result for one million elements but otherwise is clearly better.

On the Xeon (Figure 6.16), PluTo manages to beat the naive scheme again, but in contrast to the constant stencil in 2D (Figure 6.8) or the banded matrix in 2D (Figure 6.12), the speedup is much smaller. CORALS shows significantly faster execution, but the absolute speedup over the naive scheme is also smaller in comparison to the previous 2D results on the Xeon.

## 6.4 Conclusion

We have presented CORALS, a cache oblivious scheme for iterative stencil computations that performs beyond system bandwidth limitations. Even when the kernel is hardly memory bound on the Opteron, it improves the performance against the hand-optimized

naive scheme. On the Xeon where the kernel is heavily memory-bound, CORALS excels, approaching the performance of a synthetic on-chip benchmark in 2D, thus it virtually breaks the dependence on the slow off-chip connection. This is a highly desired feature, in particular, for future many-core devices that will exhibit an even larger discrepancy between the on-chip and off-chip bandwidth due to the exponential growth of CPU cores. On 3D domains, the results are less astounding but still clearly superior to the performance of the general parallelizer and locality optimizer PluTo. This is an expected result from a more specialized cache oblivious algorithm, but has not been demonstrated before.

# Part II

# Iterative Stencil Computations for Non Uniform Memory Access (NUMA) Systems

# Chapter 7

# NUMA Aware Iterative Stencil Computations on Many-Core Systems

So far the NUMA (see Section 2.3) nature of today's machines has been largely ignored in tiling schemes despite its crucial importance for scalability and the fact that the related problem of minimizing communication in a distributed memory system has been already analyzed for one of the first temporal blocking schemes by Wonnacott [72]. To systematically devise an algorithm that delivers scalable high performance results, we include the NUMA aspect as an equally important goal in our list of four key requirements for efficient temporal blocking schemes on ccNUMA machines:

- **spatio-temporal data locality.**
- **parallelization.**
- **regular memory access.**
- **data-to-core affinity.**

In this Chapter, we build upon our previous cache-aware CATS [60] and cache-oblivious CORALS [59] schemes that perform well on symmetric multiprocessing (SMP) memory systems but exhibit unsatisfactory scalability on machines with cache coherent non-uniform memory architecture (ccNUMA). Adding data-to-core affinity to these schemes is a challenge because the requirements are in conflict, e.g., parallelization conflicts with data-to-core affinity when an idle processor could process data that has been allocated by threads running on a different core. In case of CATS these conflicts can be resolved

more easily than for CORALS, which requires a new tiling and parallelization strategy and becomes a significantly different scheme than the original.

## 7.1 NUMA-aware CATS Scheme (nuCATS)

The cache-aware CATS (cache accurate time skewing) scheme [60] divides the space-time into large tiles, much larger than the cache. However, the tiles have a carefully chosen cross-section that allows a cache efficient wavefront traversal of them. The processing within the tile, i.e., the wavefront traversal, does not change in nuCATS, however, the tiling and the scheduling of the tiles changes.

CATS assigns threads to tiles in a round robin fashion, such as to reduce synchronization and obtain automatic load balancing, because tiles at the domain boundary are smaller than inside the domain. However, such an assignment violates the data-to-core affinity requirement, because a thread may be assigned a tile that resides anywhere in the domain; nuCATS performs a domain decomposition so that each thread owns a subdomain. Then it assigns tiles to threads based on which subdomain contains most of the tile. For simplicity nuCATS enforces a particularly regular pattern of how tiles and subdomains match.

Formulas inherited from CATS deliver the recommended wavefront size of tiles based on cache parameters. From this nuCATS computes the number of tiles that could fit side by side along the dimension designated for tiling. We distinguish two cases; the first case when the number of tiles is greater than but does not divide the number of threads, then we reduce the wavefront size and thus the number of tiles is enlarged until it divides the number of threads. The second case when the number of threads is greater than but does not divide the number of tiles, then similar to the first case the wavefront size is reduced and thus number of tiles is enlarged until it equals the number of threads. However, often for the second case when the number of threads is huge, this could result in a wavefront size smaller than a heuristic value computed from the cache parameters, we stop reducing the wavefront size when it is equal to half the number of threads. The number of tiles is then doubled by cutting the dimension of the wavefront traversal in half. This reduces locality, however, is still better than cutting the unit-stride dimension which would also affect the utilization of the system bandwidth. At this stage the number of tiles is equal to or a multiple of the number of threads and each thread is assigned to one or multiple tiles that lie within its subdomain.

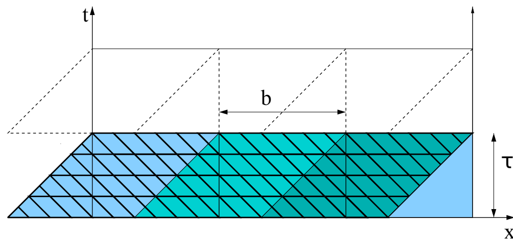



**Figure 7.1.** Bidirectional tiling. Large thread parallelograms are skewed to the right and depicted with different colors. Small base parallograms are skewed to the left. Vertical lines separate special tiles of width $b$. Each spatial tile is allocated and initialized by a different thread to assure data-to-core affinity. $\tau$ denotes the height of thread parallelograms, it parameterizes the trade-off between data-to-core affinity and temporal locality. Dotted parallelograms depict the next layer of thread parallelograms.

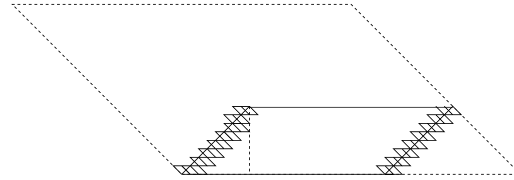


**Figure 7.2.** One thread parallelogram covered by a root parallelogram (dotted). The small base parallelograms are formed be subdivision of the root parallelogram. Threads must synchronize at base parallelograms that intersect the thread parallelogram boundary; the lower part of each intersecting base parallelogram must be computed first, before a different thread in a neighboring thread parallelogram may compute the upper part. The part to the left of the dotted vertical line has been allocated by the neighboring thread parallelogram.

## 7.2 NUMA-aware CORALS Scheme (nuCORALS)

The original cache-oblivious CORALS (cache oblivious parallelograms) scheme [59] creates a regular hierarchical space-time decomposition into parallelograms, which serves both for cache oblivious data locality and parallelization. Data-to-core affinity cannot be directly incorporated into this decomposition, therefore, the new NUMA-aware CORALS scheme (nuCORALS) inherits only the single-threaded treatment of data locality from its predecessor and creates a second level of tiling with different parallelization and synchronization. We describe the entire scheme in the following.

### 7.2.1 Bidirectional Tiling

The spatial dimensions form a tensor product and each of them relates to the time dimension in the same fashion. Therefore, to explain most aspects of nuCORALS it suffices to discuss the relation between one spatial dimension and the time dimension in a 2D space-time, see Figure 7.1 and 7.2. Section 7.2.4 discusses those properties that need additional consideration in multiple dimensions.

The scheme runs in three phases:

**Phase I**: **NUMA-aware spatial domain decomposition and data-to-core affinity maximization**

We tile the *spatial* dimensions such that the overall number of *spatial tiles* is equal to the

number of threads executing the scheme. In Figure 7.1, the spatial tiles are one dimensional with width $b$. We use affinity routines to pin each thread to one core before it allocates and initializes one spatial tile. As such, each spatial tile is allocated in the memory attached to the processor running the thread (first touch strategy) and by allowing each thread to process the spatial tile it has allocated, we ensure the data-to core affinity requirement is satisfied.

**Phase II**: **Parallelization**
We tile the temporal dimension according to the parameter $\tau$ (see Figure 7.1) into a certain number of *temporal tiles*. Section 7.2.3 discusses the selection of the parameter $\tau$ in detail. The tensor product of the spatial tiles with the temporal tiles results in layers of *space-time slices*.

To allow multiple threads to start in parallel, space-time slices are skewed to the right with a slope equal to the stencil order, resulting in parallelograms which we refer to as *thread parallelograms*. Thread parallelograms (depicted in different colors in Figure 7.1) at the left boundary of each spatial dimension are wrapped around to support periodic boundary conditions.

**Phase III**: **Cache oblivious decomposition and stencil kernel computation**
Each thread proceeds by covering its thread parallelogram by a single space-time parallelogram, which we call *root parallelogram*. Root parallelograms are skewed to the left with a slope equal to the stencil order to respect the stencil dependencies. We skew thread parallelograms to the right and root parallelograms to the left and not vice versa, because the alternative would require to process thread parallelograms from right to left, which works against the prefetcher.

nuCORALS recursively subdivides the root parallelogram into *intermediate parallelograms* striving to maximize their volume-to-surface area ratio. To this end, always the longest dimension (including time) of the intermediate parallelograms is subdivided. The subdivision is stopped when all dimensions of the current intermediate parallelograms have reached a certain size, we call the resultant parallelograms which are not subdivided further *base parallelograms*. A single-threaded kernel is then applied on the data covered by the base parallelograms. Once all threads have finished executing the kernels on the data covered by their thread parallelograms, they synchronize before they proceed to the next layer of space-time slices and execute phase III repeatedly until all layers are processed.

### 7.2.2 Synchronization

Threads are synchronized in two places, between each pair of thread parallelograms and at the boundary of each layer of space-time slices. For the latter, one could synchronize

each thread parallelogram with the two thread parallelograms beneath it. Since this synchronization does not happen very often due to the relatively small number of thread parallelograms, we use barriers in pthreads to synchronize all threads at the boundary of each layer of space time slices. We call this *global* synchronization, since all threads are involved.

Base parallelograms that intersect the boundary of any thread parallelogram (Figure 7.2) must be processed by multiple threads in a certain order. Therefore, synchronization is needed between these threads. We attach a structure of synchronization flags to each thread. Each flag represents the index of a base parallelogram within the root parallelogram space. We distinguish two checks, the first is the intersection with the right boundary of the thread parallelogram, and the second is the intersection with the left boundary. If a base parallelogram intersects the right boundary of a thread parallelogram, then the thread enters a spin-wait loop waiting for the flag of that base parallelogram to be set. If a base parallelogram intersects the left boundary of a thread parallelogram, then the thread processes all data that belongs to its thread parallelogram, i.e., the lower part of the base parallelogram, and then sets the corresponding flag in the synchronization structure of the adjacent thread whose right boundary intersects with this base parallelogram. We call this *local* synchronization.

### 7.2.3 Internal Parameters

nuCORALS has several internal parameters which are hidden from the user. Tuning these parameters can yield higher performance on some machines, however, we fix them for easier code portability of our schemes.

As is the case for most practical implementations of cache oblivious algorithms, we stop the recursive subdivision of the space-time domain once the tile is sufficiently small because going deeper in the recursion tree, until single space-time points are reached, would produce more control logic overhead than the actual computation. Furthermore, tiles with single space-time points limit the optimization opportunities for the computation inside the tiles such as innermost loop unrolling and vectorization, see [56]. We compute the dimensions of the base parallelogram in the same way as in CORALS [59].

The internal parameter $\tau$ is the height of a thread parallelogram, it represents a trade-off between temporal locality and data-to-core affinity. For stencil order $s = 1$, the ratio of data items processed by one thread but allocated by another to the overall number of items computed by this particular thread is $\tau/2b$, where $b$ is the width of the thread parallelogram which can be computed as the size of spatial dimension divided by the thread count. We can obtain more temporal locality by increasing $\tau$ at the expense of less

data-to-core affinity, because larger $\tau$ results in bigger fractions of data being processed by one thread but allocated by another. The same effect appears when $b$ becomes small, e.g., due to a high number of threads, however, we solve this problem by parallelizing across multiple dimensions, see Section 7.2.4. Through some experiments, we have found that setting $\tau = b/2$ to be half the width of the thread parallelograms results in a good trade-off between these two conflicting requirements: 75% of the overall processed data are local.

### 7.2.4 Multidimensional Properties

This section explains the properties of our scheme for an arbitrary dimension $m$ of the space-time. We explain the differences to the 2D iteration space and refer for analogy to the previous 2D figures.

**Domain decomposition.** In an $m$ dimensional space-time and $n$ threads, we create $n$ tiles by dividing all dimensions except for the unit-stride since this reduces the bandwidth utilization [15, 33]. Each dimension is subdivided into approximately $n^{1/(m-2)}$ tiles where $m-2$ results from excluding the time and the unit-stride dimensions. If $n^{1/(m-2)}$ is not an integer, we favor dimensions with a higher stride, e.g., for $m = 4$D space-time domain and $n = 4$, only two dimensions are subdivided, each dimension is subdivided into 2 tiles; for $n = 8$, the dimension with highest stride is subdivided into 4 tiles and the other is subdivided into 2 tiles.

**Synchronization.** Synchronization is similar to the 2D case, the only difference is that local synchronization is now needed between each adjacent pair of thread parallelograms in each dimension. This results in more checks for intersection with the left or the right boundaries of the thread parallelogram in each dimension. However, these checks are cheap and hardly impact the running time of the scheme.

**Internal parameters.** In 2D we have $\tau = b/2$, where $b$ is the width of the thread parallelograms. For higher dimensional space-time, we use the same formula only $b$ is now the smallest spatial dimension of the thread parallelograms. The domain decomposition tries to tile the spatial dimensions equally so that $\tau$ becomes as large as possible without degrading data-to-core affinity.

## 7.3 Results

### 7.3.1 Schemes

In the experiments the following schemes are compared:

- **PeakDP**: Measured computational peak in double precision. We obtain this value by performing a sequence of independent multiply-add operations in registers. PeakDP models the absolute upper bound for any computation on a machine. It is clear that no optimization of stencil codes will reach this upper bound since stencil operations are not independent. The goal is to achieve a high fraction of this peak.

- **LL1Band0C**: Last-level cache bandwidth with zero further caching. It models the performance of a stencil code in case the domain could entirely fit into the last-level cache, but no higher level caches are present. Accordingly, for the case of a 7-point constant stencil of order $s = 1$, 7 read and 1 write operations are performed from the last-level cache for each kernel execution. For the variable stencil case (banded-matrix), 14 reads (7 vector elements plus 7 matrix coefficients) and 1 write operations are counted. LL1Band0C represents the achievable performance in case of an enormous last level cache that could hold all data on-chip.

- **nuCATS**: Our NUMA-aware, cache-aware scheme from Section 7.1; nuCATS is parallelized with pthreads and the kernel is vectorized using SSE2 intrinsics to prevent it from becoming compute-bound.

- **nuCORALS**: Our NUMA-aware, cache oblivious scheme from Section 7.2; nuCORALS is parallelized with pthreads and the kernel is vectorized using SSE2 intrinsics to prevent it from becoming compute-bound.

- **CATS**: Our original cache aware time skewing scheme [60] (Chapter 4).

- **CORALS**: Our original cache oblivious parallelograms scheme [59] (Chapter 6).

- **Pochoir**: Code compiled using Phase II compilation of the Pochoir compiler and run-time system for implementing stencil computations on multicore processors [63]. We modify the kernel function of the 3D 7-point stencil example provided in the examples folder of the Pochoir package to implement (7.1) and use Pochoir's latest version v0.5 to compile it with the `-O3 -ipo -xHost` flags. Other flags suggested in the makefile either do not affect or worsen the performance.

- **PLuTo** code transformed by the automatic parallelizer and locality optimizer for multicores PLuTo version 0.7.0 [7]. We have tuned the tile sizes for our machines individually and use the transformation flags that yield the best performance. The transformed code is compiled with intel icc compiler version 12.1.2 with the `-O3 -ipo -openmp -parallel` flags and it reports successful vectorization of the loops.

- **SysBandIC**: System bandwidth with ideal caching.
  A performance estimate derived from the measured peak system bandwidth, see Table 7.1. It assumes a sufficiently large cache that can hold at least 2 slices of the

3D domain or 2 lines of the 2D domain; therefore, for the case of 7-point constant stencil of order $s = 1$, 1 read and 1 write operations are performed from main memory for each kernel execution. For the variable stencil case (banded-matrix), 8 reads (7 vector elements plus 7 matrix coefficients) and 1 write operations are counted. SysBandIC models the absolute upper bound for the performance of a naive implementation of stencil codes when the domain is too big to fit entirely into the cache.

- **NaiveSSE**: A Naive implementation with the following optimizations employed: parallelization using pthreads, kernel vectorization using SSE2 intrinsics, and NUMA-aware data allocation. We expect that the NaiveSSE curve will lie between Sys-Band0C and SysBandIC.

- **SysBand0C**: System bandwidth with zero-caching.
  In contrast to SysBandIC, it assumes there is no cache and thus all data accesses go to main memory. For the case of 7-point constant stencil of order $s = 1$, 7 read and 1 write operations are performed from main memory for each kernel execution. For the variable stencil case (banded-matrix), 14 reads (7 vector elements plus 7 matrix coefficients) and 1 write operations are counted. SysBand0C represents the lower bound for the performance of an efficient naive implementation of stencil codes.

The LL1Band0C, SysBandIC, and SysBand0C schemes assume that the bandwidth is the sole limiting factor, and all other factors (memory access latency, access to higher level memories, computation, etc.) are hidden behind it. Due to layout restrictions we refer in the figures to the suffix 'Band' with only the letter 'B'.

### 7.3.2 Testbed

Our testbed comprises constant and variable (banded matrix) 7-point stencils with order $s = 1$. Each stencil execution performs 7 multiplications and 6 additions amounting to 13 flops. A single stencil point update in 3D is described by

$$\begin{aligned} X_{i,j,k}^{t+1} = c_1 \cdot X_{i-1,j,k}^{t} + c_2 \cdot X_{i,j-1,k}^{t} + c_3 \cdot X_{i,j,k-1}^{t} \\ + c_4 \cdot X_{i+1,j,k}^{t} + c_5 \cdot X_{i,j+1,k}^{t} + c_6 \cdot X_{i,j,k+1}^{t} \\ + c_0 \cdot X_{i,j,k}^{t} \end{aligned} \quad (7.1)$$

where $c_i$, for $0 \leq i \leq 6$ are the stencil coefficients.

We show both weak and strong scalability of nuCATS, nuCORALS and the other schemes on the two machines whose specifications are listed in Table 7.1. To prevent the early

exploitation of another socket's system bandwidth before all cores on one socket are in use, we use the affinity routines to pin the thread contexts to cores on one socket, before occupying a new socket.

We demonstrate the weak scalability of nuCATS and nuCORALS on a $200^3$ domain per core configuration, whereby the domain on which we compute in case of $n$ threads is not an agglomeration of $n$ separate $200^3$ cubes, but one cube of volume $n \cdot 200^3$. Thus, with growing thread number, the weak scalability is not trivial, as it becomes more and more difficult to exploit data locality in the large data cubes. The strong scalability is presented for $160^3$ and $500^3$ domains. In the $160^3$ case, the challenge is the shrinking working size for each thread that makes the inter-core communication become a bigger relative overhead. Not surprisingly it is therefore easier to obtain good scalability on the large $500^3$ domain.

We run 100 iterations with two copies of $X$ instead of in-place updates of Gauss-Seidel type with one one copy of $X$, since the two copy scenario is more general and challenging for temporal blocking. Temporal blocking is also beneficial for fewer iterations, e.g., to accelerate multiple smoother applications on each level of a multigrid solver, however, for a general performance comparison of temporal blocking schemes 100 iterations are more suitable.

All figures show the number of cores involved in executing the schemes on the x-axis. Figure 7.4 to Figure 7.15 have two y-axes; the left one shows how many giga updates of $X^{t+1}$ can be executed per second (Gupdates/s) per core, and the right one shows the achieved GFLOPS per core with stencil (7.1), i.e., Gupdates/s times 13 in this case. We show Gupdates/s since it is a more informative measure when the performance of different stencils is compared, e.g., Gupdates/s hardly changes if we add another stencil point to (7.1) because the problem is still memory bound, however, the GLOPS number would change immediately. Since all graphs show results per core, a straight horizontal line means linear scaling with the number of cores.

### 7.3.3 Memory Bandwidth

Figure 7.3 shows how memory and cache bandwidths scale with the number of cores. For both machines, the cache bandwidth scales linearly with the number of cores, because each core has its own connection to the caches.

One thread does not saturate the memory bus. For the Opteron 8222 machine, the single-threaded memory bandwidth grows by a factor of 1.6x when 2 cores are used. The bandwidth increases on average by a factor of 1.5x when one additional socket is used up to 8 sockets (number of sockets in the machine). Overall, Opteron's single threaded bandwidth grows by 6.5x when all 16 cores on all sockets are employed.

**Table 7.1.** Hardware configurations. The measured bandwidth numbers come from the STREAM COPY benchmark tool running with 16 and 32 threads and SSE reads. The measured peak double precision (DP) FLOPS come from our SSE benchmark consisting of independent multiply-add operations on registers. LL1 denotes last-level cache, LL2 denotes last but one level cache.

| Brand | AMD | Intel |
|---|---|---|
| Processor | Opteron 8222 | Xeon X7550 |
| Code-named | Santa Rosa | Beckton |
| Frequency | 3.0 GHz | 2.0 GHz |
| Number of sockets | 8 | 4 |
| Cores per socket | 2 | 8 |
| L1 Cache per core | 64 KiB | 32 KiB |
| L2 Cache per core | 1 MiB | 256 KiB |
| L3 Cache per core | - | 2.25 MiB |
| Operating system | Linux 64 bit | Linux 64 bit |
| Parallelization | 1..16 pthreads | 1..32 pthreads |
| Vectorization | SSE2 | SSE2 |
| Number of NUMA Nodes | 8 | 4 |
| Compiler | g++ 4.3.2 | icpc 12.1.2 |
| Measured L1 Bandwidth | 675.3 GB/s | 819.1 GB/s |
| Measured L2 Bandwidth | 185.7 GB/s | 642.8 GB/s |
| Measured L3 Bandwidth | - | 588.6 GB/s |
| Measured Sys. Bandwidth | 11.9 GB/s | 63.0 GB/s |
| Measured Peak DP FLOPS | 95.3 G | 202.5 G |
| LL1 Band./Sys. Bandwidth | 15.6 | 9.3 |
| LL2 Band./LL1. Band. | 3.6 | 1.1 |
| Peak DP/(Sys. Band./8B) Arith. intensity for Sys. | 64.1 | 25.7 |
| Peak DP/(LL1 Band./8B) Arith. intensity for LL1 | 4.1 | 2.8 |

For the the Xeon X7550 machine, memory bandwidth scales almost linearly from 1 to 2 cores; from 2 to 4 cores, bandwidth grows by 1.7x. Using all 8 cores on one socket saturates the bus since bandwidth increases by only 1.5x. Bandwidth grows by a factor of 1.4x when another socket is used. Overall, Xeon's single threaded bandwidth grows by a factor of 13.7x when all cores on the four sockets are engaged.

The cache and bandwidth performance numbers displayed in Figure 7.3 are used to define the benchmarks LL1Band0C, SysBandIC and SysBand0C based on the *Roofline model* [69]. Clearly all schemes and benchmarks will achieve higher performance on the Xeon than on the Opteron due to the higher cache and memory bandwidths. However, the system bandwidth per core goes down significantly in both cases. So to obtain linear scalability with a temporal blocking scheme, the scheme has to create so much temporal locality and so few cache misses, that its scalability starts depending mostly on the linear scalability of the cache bandwidth rather than the degrading scalability of the system

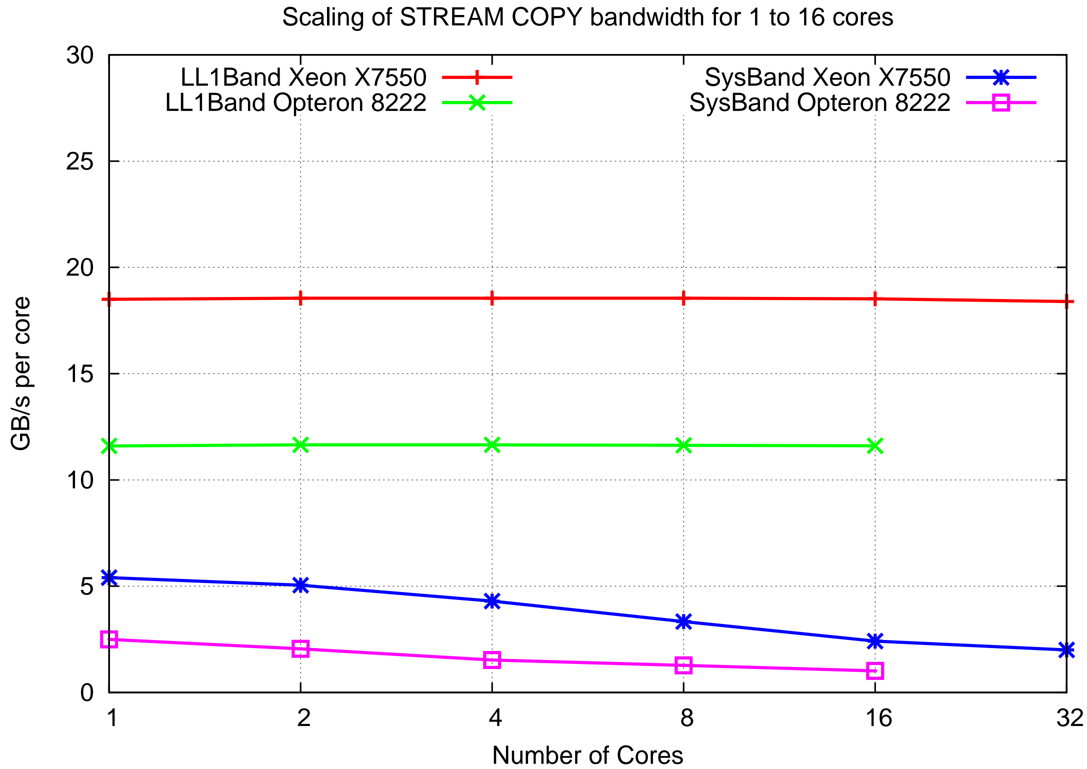



**Figure 7.3.** Scalability of last-level cache and system bandwidths for 1 to 16 threads on Opteron 8222 machine and for 1 to 32 threads on Xeon X7550 machine.

bandwidth. We will see that nuCATS and nuCORALS cannot decouple completely from the degrading scalability of the system bandwidth, however, in most case the scalability is much better.

### 7.3.4 Scalability for Constant Stencils

From Figure 7.4 to Figure 7.9, we can draw the following common conclusions:

- **NaiveSSE, SysBandIC and SysBand0C on the Xeon are faster than their counterparts on the Opteron.**
  The performance of these schemes depends on the system bandwidth. With 16 threads (2 sockets), the Xeon has 38.7 GB/s system bandwidth while the Opteron has only 11.9 GB/s, a ratio of 3.3, see Table 7.1, and in fact NaiveSSE on the Xeon achieves a similar speedup factor of 2.7x over NaiveSSE on the Opteron.

- **LL1Band0C on the Xeon is faster than on the Opteron.**
  The performance of this benchmark depends on the bandwidth of the last cache level. The LL1 cache bandwidth on the Xeon is faster than on the Opteron, see Table 7.1 and Figure 7.3.

- **The Xeon is much faster than the Opteron on nuCORALS and nuCATS schemes.**
  The performance of nuCATS and nuCORALS depends primarily on the cache bandwidth. The Xeon X7550 features larger and faster caches than the Opteron 2218. Both schemes exploit them effectively to reduce the impact of the slow accesses to the main memory.

- **The performance graph of NaiveSSE lies between SysBandIC and SysBand0C on both machines.**
  NaiveSSE scheme performs better than SysBand0C since SysBand0C assumes that 7 vector elements are fetched from main memory for each kernel execution, whereas in reality some of them are cached. SysBandIC on the other hand performs better than NaiveSSE since SysBandIC assumes ideal caching wherein only 2 memory transactions per update are necessary and additional overhead in the real execution is not considered.

- **Although LL1Band0C transfers 4x more data than SysBand0C, it shows higher performance.**
  This is not an inherent property of LL1Band0C vs. SysBandIC. For the 7-point stencil of order $s = 1$, SysBandIC, which assumes ideal spatial blocking, reads 1 double and writes 1 double, LL1Band0C, which assumes zero further caching, reads 7 doubles and writes 1 double. The ratio of transferred data by LL1Band0C to transferred data by SysBandIC is 4 which is far less than the ratios of last-level cache bandwidth to system bandwidth 15.6 and 9.3 for the Opteron and the Xeon, respectively. However, for high order stencils, the ratio of transferred data becomes larger and may yield that SysBandIC surpasses LL1Band0C.

- **nuCATS is better on the large domains, nuCORALS is better on the small domain.**
  nuCATS diverts almost all effort towards the maximal cache reuse in the last level cache at the expense of all other optimizations. If the domain is much bigger than the last level cache this strategy pays off as it minimizes main memory traffic which is 15.6x or 9.3x slower than the last level cache bandwidth on the Opteron and the Xeon, respectively. However, on smaller domains less aggressive last level cache optimization in nuCORALS also reduces main memory traffic to a small amount, and then its additional higher level cache optimization leads to better performance.

While the schemes show various common characteristics on the two machines used for our experiments, it is not surprising that the schemes also exhibit different behavior on those machines. This is simply due to the difference in the architecture and the other

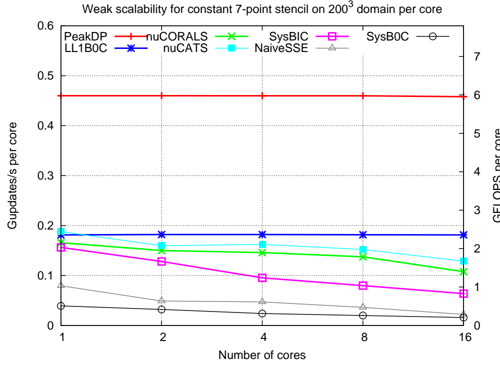



**Figure 7.4.** Constant stencil weak scalability for 1 to 16 threads with $200^3$ doubles per thread and 100 timesteps on the Opteron 8222. GFLOPS achieved with 16 cores, PeakDP 95.3, LL1Band0C 37.7, nuCORALS 22.4, nuCATS 26.8, SysBandIC 13.2, NaiveSSE 4.6, SysBand0C 3.3

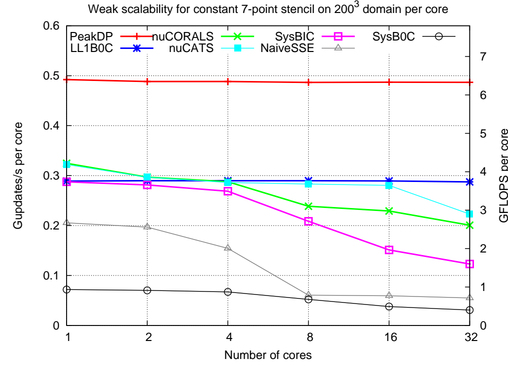



**Figure 7.5.** Constant stencil weak scalability for 1 to 32 threads with $200^3$ doubles per thread and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, PeakDP 202.5, LL1Band0C 119.6, nuCORALS 83.4, nuCATS 92.7, SysBandIC 51.2, NaiveSSE 22.9, SysBand0C 12.7

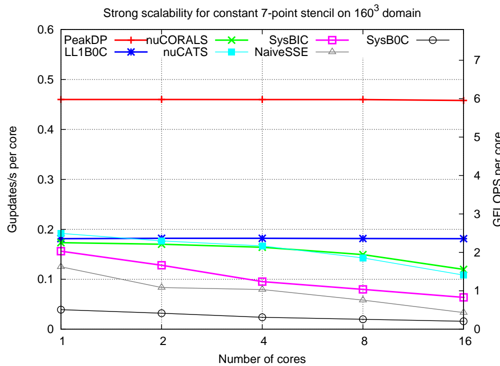



**Figure 7.6.** Constant stencil strong scalability for 1 to 16 threads on a $160^3$ domain of doubles and 100 timesteps on the Opteron 8222. GFLOPS achieved with 16 cores, PeakDP 95.3, LL1Band0C 37.7, nuCORALS 24.9, nuCATS 22.5, SysBandIC 13.2, NaiveSSE 6.9, SysBand0C 3.3

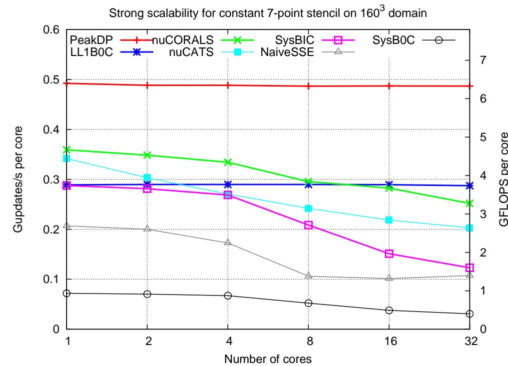



**Figure 7.7.** Constant stencil strong scalability for 1 to 32 threads on a $160^3$ domain of doubles and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, PeakDP 202.5, LL1Band0C 119.6, nuCORALS 104.8, nuCATS 84.5, SysBandIC 51.2, NaiveSSE 44.7, SysBand0C 12.7

characteristics of the machines, see Table 7.1. In the following, we explain the platform specific behavior of the schemes.

#### 7.3.4.1 Opteron Results

Figures 7.4, 7.6, and 7.8 show that the performance curves of nuCATS and nuCORALS lie between SysBandIC and LL1Band0C on the Opteron. Being faster than SysBandIC means that both schemes transfer on average less than 2 doubles from main memory per stencil update due to the created space-time data locality. Despite the degrading system bandwidth both schemes show very good scalability up to 8 cores (nearly horizontal

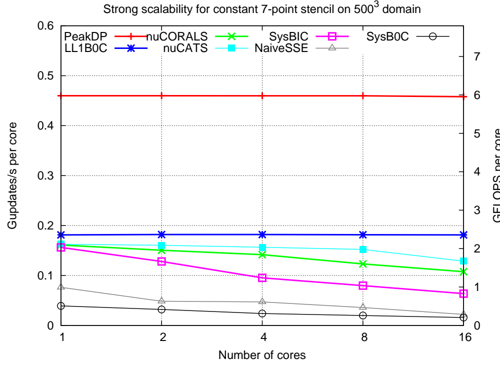



**Figure 7.8.** Constant stencil strong scalability for 1 to 16 threads on a $500^3$ domain of doubles and 100 timesteps on the Opteron 8222. GFLOPS achieved with 16 cores, PeakDP 95.3, LL1Band0C 37.7, nuCORALS 22.4, nuCATS 26.8, SysBandIC 13.2, NaiveSSE 4.6, SysBand0C 3.3

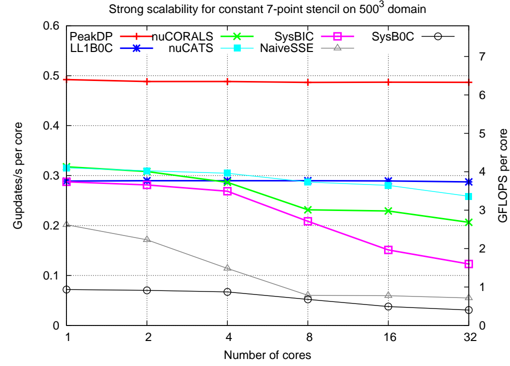



**Figure 7.9.** Constant stencil strong scalability for 1 to 32 threads on a $500^3$ domain of doubles and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, PeakDP 202.5, LL1Band0C 119.6, nuCORALS 85.9, nuCATS 107.6, SysBandIC 51.2, NaiveSSE 22.9, SysBand0C 12.7

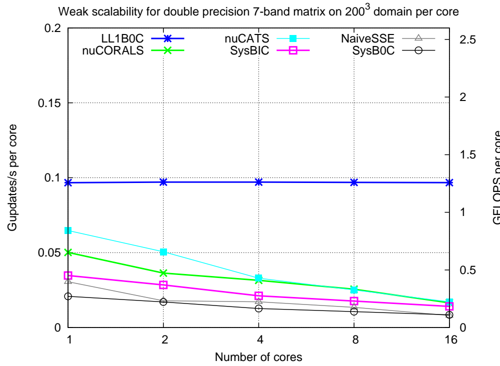



**Figure 7.10.** Banded matrix weak scalability for 1 to 16 threads with $200^3$ doubles per thread and 100 timesteps on the Opteron 8222. GFLOPS achieved with 16 cores, PeakDP 95.3, LL1Band0C 20.1, nuCORALS 3.4, nuCATS 3.6, SysBandIC 2.9, NaiveSSE 1.7, SysBand0C 1.8

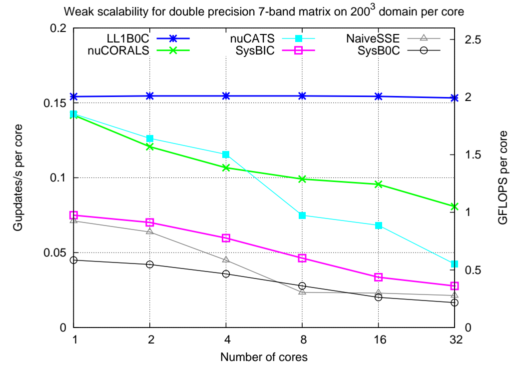



**Figure 7.11.** Banded matrix weak scalability for 1 to 32 threads with $200^3$ doubles per thread and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, PeakDP 202.5, LL1Band0C 63.8, nuCORALS 33.6, nuCATS 17.7, SysBandIC 11.3, NaiveSSE 8.9, SysBand0C 6.8

lines). When using all 16 cores of the machine, they become more affected by the system bandwidth limit and the 8 core performance of nuCORALS and nuCATS grows only by a factor of 1.6x and 1.7x.

Overall, Opteron's single-core performance on nuCORALS grows by a factor of 10.4x when using all 16 cores of the machine in Figure 7.4 ($200^3$ per core domain), 11.1x in Figure 7.6 ($160^3$ domain), and 10.7x in Figure 7.8 ($500^3$ domain). The highest fraction of the computational peak on 16 cores is reached in Figure 7.6, namely 26%. Opteron's single-core performance on nuCATS grows by a factor of 11.2x when using all 16 cores of the machine in Figure 7.4 ($200^3$ per core domain), 9.4x in Figure 7.6 ($160^3$ domain), and 11.2x in Figure 7.8 ($500^3$ domain). nuCATS achieves the highest fraction of the computational peak (28%) using all 16 cores of the machine.

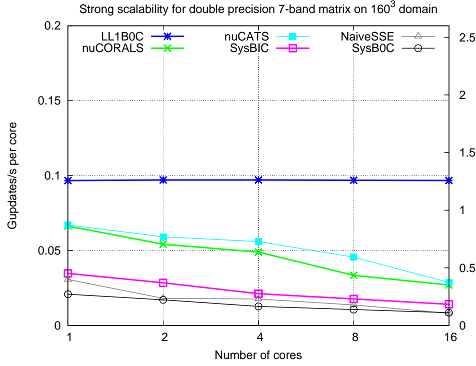



**Figure 7.12.** Banded matrix strong scalability for 1 to 16 threads on a $160^3$ domain of doubles and 100 timesteps on the Opteron 8222. GFLOPS achieved with 16 cores, PeakDP 95.3, LL1Band0C 20.1, nuCORALS 5.6, nuCATS 6.0, SysBandIC 2.9, NaiveSSE 1.7, SysBand0C 1.8

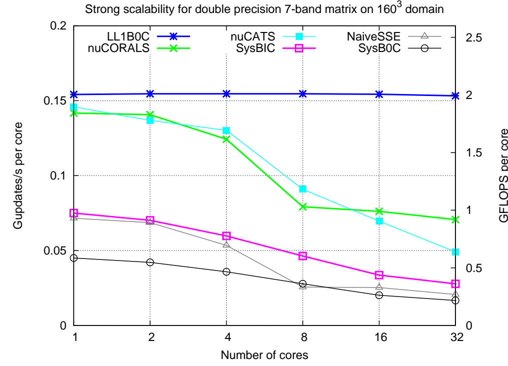



**Figure 7.13.** Banded matrix strong scalability for 1 to 32 threads on a $160^3$ domain of doubles and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, PeakDP 202.5, LL1Band0C 63.8, nuCORALS 29.4, nuCATS 20.4, SysBandIC 11.3, NaiveSSE 8.6, SysBand0C 6.8

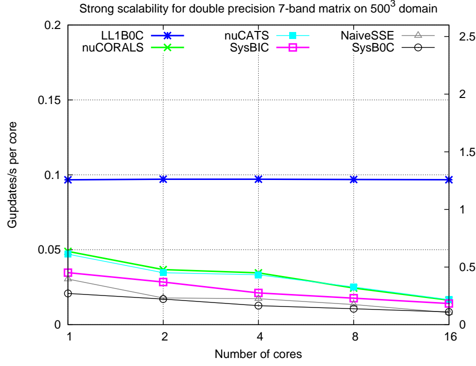



**Figure 7.14.** Banded matrix strong scalability for 1 to 16 threads on a $500^3$ domain of doubles and 100 timesteps on the Opteron 8222. GFLOPS achieved with 16 cores, PeakDP 95.3, LL1Band0C 20.1, nuCORALS 3.4, nuCATS 3.5, SysBandIC 2.9, NaiveSSE 1.7, SysBand0C 1.8

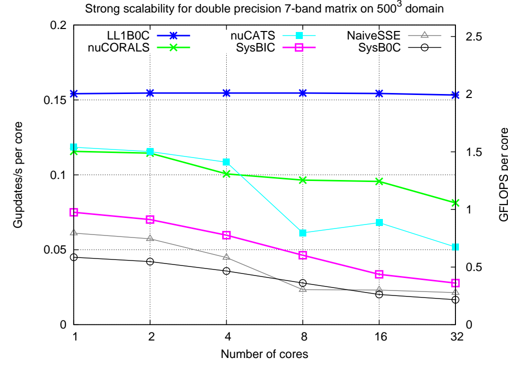



**Figure 7.15.** Banded matrix strong scalability for 1 to 32 threads on a $500^3$ domain of doubles and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, PeakDP 202.5, LL1Band0C 63.8, nuCORALS 33.8, nuCATS 21.6, SysBandIC 11.3, NaiveSSE 8.9, SysBand0C 6.8

#### 7.3.4.2 Xeon Results

On the Xeon, nuCORALS not only surpasses SysBandIC, but also it beats the performance of LL1Band0C up to 4 cores. This means that even if gigabyte large domains could fit into the last level cache and would be processed completely on-chip, the already available performance of nuCORALS is still superior. This is a remarkable result, as it shows that a cache oblivious algorithm can draw so much benefit from higher level caches that it overcompensates for the remaining slow data accesses to main memory and performs on average better than the last level cache alone. However, for higher core counts than 4, the sublinear scaling of the main memory bandwidth renders it more and more difficult to beat LL1Band0C. Only on the $160^3$ domain nuCORALS is still better than LL1Band0C with 8 cores. This is due to the big fraction of data cached in higher level caches and less

main memory traffic compared to big domains, which compensates for the increasingly slower transfers from main memory.

nuCATS optimizes for the last level cache exclusively and in fact on the large domains it shows very similar performance to LL1Band0C up to 16 cores, only for 32 cores it falls off a bit. This is a big achievement, demonstrating an algorithmic decoupling from the slow main memory bandwidth, which is already severely degrading up to 16 cores, see Figure 7.3. At first it appears very surprising that nuCATS can even beat LL1Band0C in some cases, as it has some overheads and does not optimize for anything else than the last level cache. However, similar to the processing pattern of the naive scheme, there is some natural data reuse in higher level caches.

As already discussed nuCORALS is clearly better than nuCATS on the $160^3$ domain, because the small domain allows high data reuse in higher level caches and the cache-oblivious nuCORALS automatically takes advantage of that.

Figures 7.5, 7.7, and 7.9 show that nuCORALS's per core performance falls off from 2 to 8 cores, because more and more threads compete for the shared last level cache. Despite the decreasing cache capacity per thread and the decreasing main memory bandwidth available to each thread (Figure 7.3) the drop in per-core performance is moderate. When additional sockets come into use, i.e., the transitions from 8 to $2 \cdot 8$ to $4 \cdot 8$ cores, nuCORALS maintains a near linear scalability. When all cores on all sockets are in use, nuCORALS achieves 52% of the measured computational peak performance. Overall, the Xeon's single-core performance grows on average by a factor of 22.0x when nuCORALS uses all 32 cores of the machine and by a factor of 22.7x when nuCATS uses all 32 cores of the machine.

To summarize, nuCATS and nuCORALS perform very well on both the Xeon and the Opteron. They show a near linear scalability where the system bandwidth scales almost linearly and still good scalability where system bandwidth scales only sublinearly.

### 7.3.5 Scalability for Banded Matrices

Common to all banded matrix figures is the omission of PeakDP (Section 7.3.1), because its inclusion would severely compress all other graphs at the bottom. However, it is important to keep in mind that PeakDP is much higher than the displayed LL1Band0C and represents the real extent of the memory wall problem.

When the stencil coefficients are not constant, they must be stored in main memory. This corresponds to a banded matrix vector product. In this case, to exploit temporal locality, not only vector elements, but also the coefficients must reside in cache. Therefore, another 7 components along with each vector value must be fetched from main memory. This makes

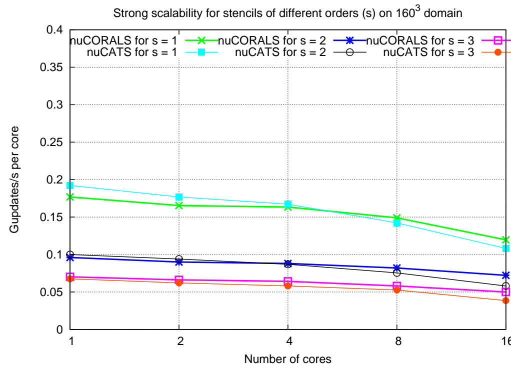



**Figure 7.16.** Strong scalability for high order stencils on a $160^3$ domain of doubles and 100 timesteps on the Opteron 8222. GFLOPS achieved for $s = 1$ nuCORALS 24.9, nuCATS 22.5. For $s = 2$ nuCORALS 28.9, nuCATS 23.2. For $s = 3$ nuCORALS 29.6, nuCATS 22.8

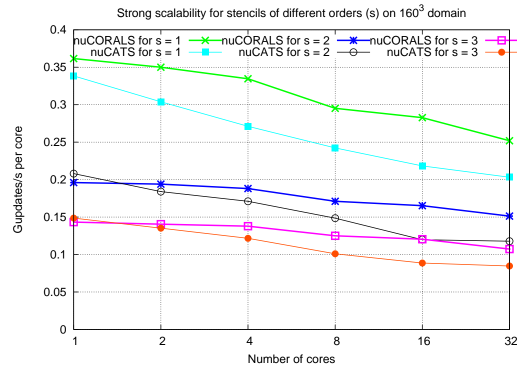



**Figure 7.17.** Strong scalability for high order stencils on a $160^3$ domain of doubles and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores for $s = 1$ nuCORALS 104.8, nuCATS 84.5. For $s = 2$ nuCORALS 121, nuCATS 94.2. For $s = 3$ nuCORALS 127, nuCATS 100.3

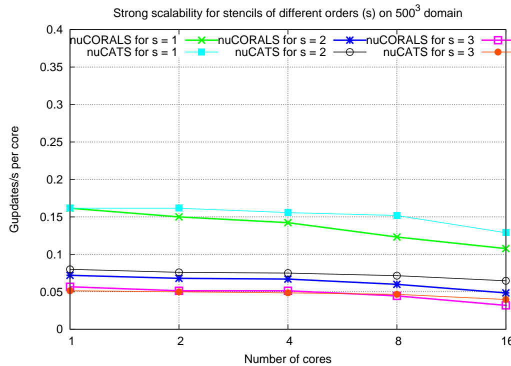



**Figure 7.18.** Strong scalability for high order stencils on a $500^3$ domain of doubles and 100 timesteps on the Opteron 8222. GFLOPS achieved for $s = 1$ nuCORALS 22.4, nuCATS 26.8. For $s = 2$ nuCORALS 19.4, nuCATS 25.9. For $s = 3$ nuCORALS 18.9, nuCATS 23.5

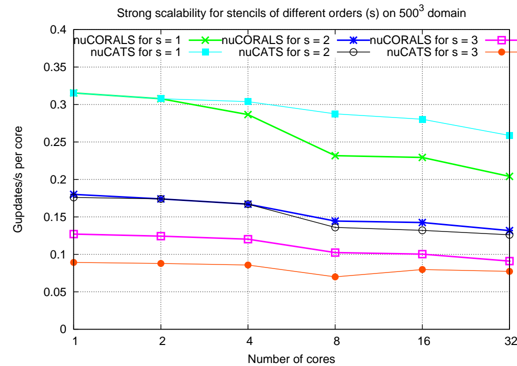



**Figure 7.19.** Strong scalability for high order stencils on a 5003 domain of doubles and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores for $s = 1$ nuCORALS 85.9, nuCATS 107.6. For $s = 2$ nuCORALS 105.4, nuCATS 100.9. For $s = 3$ nuCORALS 107.7, nuCATS 91.5

the problem even more memory-bound. When all 16 cores are used, nuCORALS's and nuCATS's aggregate performances drop by a factor of 6.6x and 7.6x, respectively, on both the $200^3$ per core and $500^3$ domains compared to the constant stencil case on the Opteron. Xeon's big L3 cache and relatively high system memory bandwidth (Table 7.1) are able to mitigate the problem to some extent and therefore, its aggregate performance drops by a factor of only 3x for nuCORALS and 5x for nuCATS.

On the Opteron, the additional data transfers create a large gap between nuCORALS and nuCATS on the one side and LL1Band0C on the other side. Both schemes maintain a clear advantage over SysBandIC, however, the additional main memory traffic makes them also inherit its sublinear scalability. The single-threaded performance of nuCORALS and nuCATS accelerates by around 6x on the $160^3$ (Figure 7.12) and $500^3$ (Figure 7.14)

domains and around 5x on the $200^3$ per core problem (Figure 7.10), when all 16 cores are engaged in the computation. The latter is particularly difficult to accelerate because the dependence on the system bus grows super-linearly (linear in volume plus more tile boundaries in large volume), while system bandwidth per thread decreases.

On the Xeon, the additional transfer of matrix coefficients prevents nuCORALS from surpassing the performance of LL1Band0C as in the constant stencil case. Benefiting from the large shared last-level cache, the single-threaded performance of nuCORALS and nuCATS is much closer to LL1Band0C than on the Opteron. However, it falls off rapidly when more cores are engaged in the computation, because the advantage of the shared cache disappears when it has to be divided among all cores on the same socket (There is almost no data reuse between tiles of different threads). The corresponding reduction in per-core performance is particularly strong for nuCATS and the 4 to 8 core transition, because the available last-level cache capacity per thread is halved and system bandwidth scales particularly poorly for this transition, see Figure 7.3. Although the reduction from 4 to 8 cores is disproportionally large, on average nuCATS's performance correlates with SysBandIC. nuCORALS suffers a similar reduction in per-core performance on the $160^3$ domain (Figures 7.13), however, maintains per-core performance on the bigger domains in Figures 7.13 and 7.15. The cache oblivious nature of the algorithm with the automatic exploitation of the entire cache hierarchy is of great help in these cases.

So nuCORALS is the clear winner against nuCATS for the banded matrix multiplication. It maintains more than 50% parallel efficiency on all domains, achieving speedups of 18.7x on the $200^3$ per core domain, 16.3x on the $160^3$ domain, and 22.5x on the $500^3$ domain with 32 threads. nuCATS's per core performance is 9.3x higher than its single-threaded performance on the $200^3$ per core domain, 11.3x on the $160^3$ domain, and 14.4x on the $500^3$ domain.

### 7.3.6 Scalability for High Order Stencils

Skewing thread and root parallelograms with a slope equal to the stencil order $s$ makes it more challenging to achieve high performance and scalable results. We have more control overhead from additional boundary intersections and synchronizations, the tiles' surface to volume ratios increase and more surface layers must be kept on-chip, and a larger fraction of data is processed by one thread but owned by another in case of a fixed thread parallelogram height ($\tau$ in Figure 7.1). The last effect can be alleviated by setting $\tau = b/(2 \cdot s)$, which recovers the previous compromise between data-to-core affinity and temporal blocking.

Figures 7.16 to 7.19 show the scalability of nuCORALS for stencil orders $s = 1$, $s = 2$, and

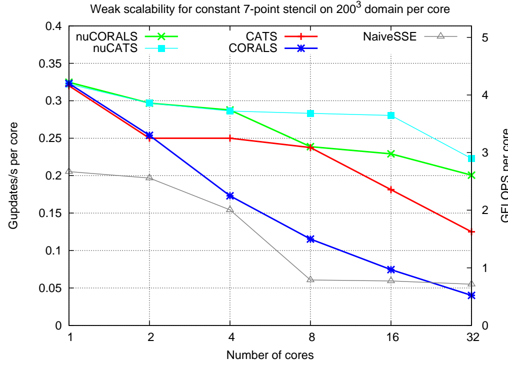



**Figure 7.20.** Constant stencil weak scalability for 1 to 32 threads with $200^3$ doubles per thread and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, nuCORALS 83.4, nuCATS 92.7, CATS 52, CORALS 16.7, NaiveSSE 22.9

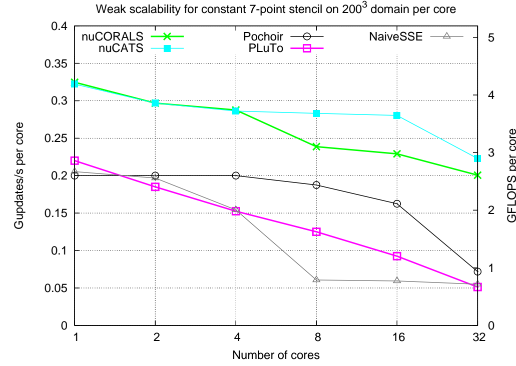



**Figure 7.21.** Constant stencil weak scalability for 1 to 32 threads with $200^3$ doubles per thread and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, nuCORALS 83.4, nuCATS 92.7, Pochoir 29.9, PLuTo 21.3, NaiveSSE 22.9

$s = 3$. Our model problem has 25 flops for $s = 2$ (13 multiplications and 12 additions), and 37 flops for $s = 3$ (19 multiplications and 18 additions). The scalability behavior of nuCORALS and nuCATS for $s = 2$ and 3 is not much different compared to the $s = 1$ case discussed above. The absolute performance clearly decreases, however, as a very positive result we observe that the decrease from $s = 1$ to $s = 2$ is less than 2x, and from $s = 1$ to $s = 3$ less than 3x, although the convex hull of the stencil required for spatial locality on-chip grows cubically.

### 7.3.7 Performance Comparison

This section highlights the importance of data-to-core affinity by comparing the performance of nuCORALS and nuCATS with CATS (Chapter 4) and CORALS (Chapter 6). We also compare against other recent temporal blocking schemes from literature: PLuTo 0.7.0 [7] and Pochoir 0.5 [63]. All schemes but nuCORALS, nuCATS, and NaiveSSE do not explicitly pay attention to this requirement; therefore, we anticipate that they will exhibit worse scalability beyond one NUMA node.

Figures 7.20, 7.24, and 7.22 show that the performances of CORALS and CATS are on par with their NUMA-aware counterparts using one core since each scheme is similar to its NUMA-aware counterpart. However, when up to 8 cores (one socket) are engaged in the computation, the graphs of CORALS vs. nuCORALS and CATS vs. nuCATS already drift apart, although both are still running on the same NUMA node. The per-thread local data allocation in nuCATS and nuCORALS helps also the efficient utilization of multi-channel memory buses. The difference between CATS and nuCATS is smaller than between CORALS and nuCORALS, because nuCORALS underwent more significant changes including a second level tiling for more coarse-granular parallelization and

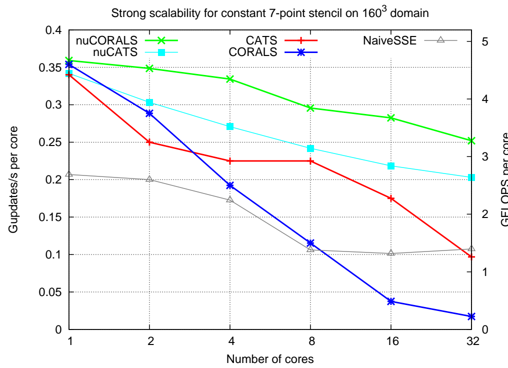



**Figure 7.22.** Constant stencil strong scalability for 1 to 32 threads on a $160^3$ domain of doubles and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, nuCORALS 104.8, nuCATS 84.5, CATS 40.3, CORALS 7.2, NaiveSSE 44.7

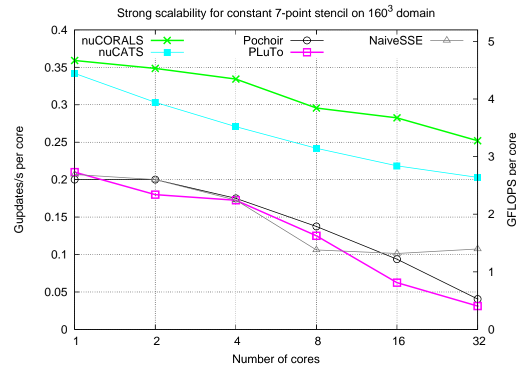



**Figure 7.23.** Constant stencil strong scalability for 1 to 32 threads on a $160^3$ domain of doubles and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, nuCORALS 104.8, nuCATS 84.5, Pochoir 16.9, PLuTo 13, NaiveSSE 44.7

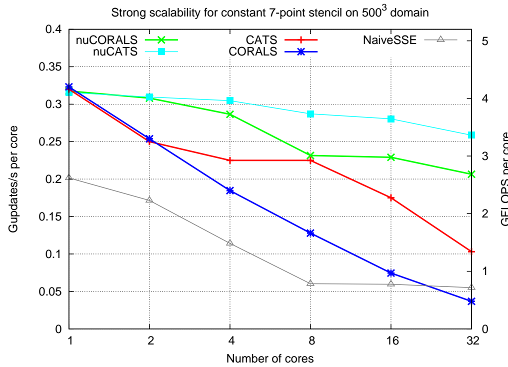



**Figure 7.24.** Constant stencil strong scalability for 1 to 32 threads on a $500^3$ domain of doubles and 100 timesteps on the Opteron X7550. GFLOPS achieved with 32 cores, nuCORALS 85.9, nuCATS 107.6, CATS 42.9, CORALS 15.3, NaiveSSE 22.9

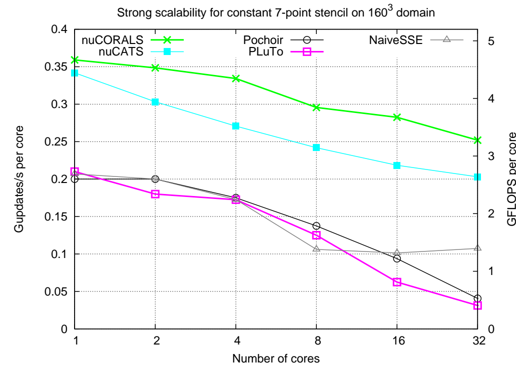



**Figure 7.25.** Constant stencil strong scalability for 1 to 32 threads on a $500^3$ domain of doubles and 100 timesteps on the Opteron X7550. GFLOPS achieved with 32 cores, nuCORALS 85.9, nuCATS 107.6, Pochoir 27.3, PLuTo 22.1, NaiveSSE 22.9

synchronization.

The NUMA importance is underlined when more than 8 cores are engaged in the computation. All non-NUMA-aware schemes suffer a big slowdown in the per-core metric as the computation goes beyond one NUMA node; nuCATS and nuCORALS on the other hand maintain a high, rather stable per-core performance level. Pochoir is quite stable up to 8 cores and then drops off sharply, while PLuTo's per-core performance degrades gradually with the number of cores.

Figures 7.22 and 7.25 report strong scaling on a rather small domain and shows particularly dramatic performance degradation on all schemes that do not observe the data-to-core affinity. For 32 cores the naive scheme is clearly faster (more than 2.5x) than all non-NUMA-aware temporal blocking schemes apart from CATS, which is only slightly worse;

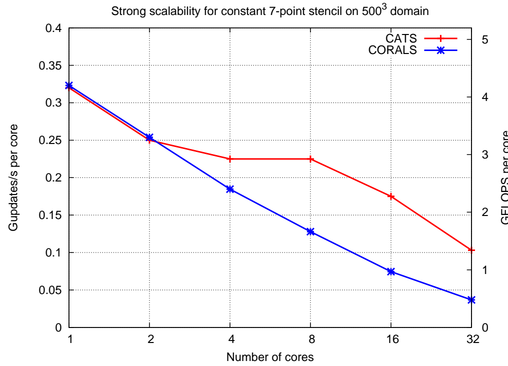



**Figure 7.26.** Constant stencil strong scalability for 1 to 32 threads on a $500^3$ domain of doubles and 100 timesteps on the Opteron X7550. GFLOPS achieved with 32 cores, nuCORALS 85.9, nuCATS 107.6, CATS 42.9, CORALS 15.3, Pochoir 27.3, PLuTo 22.1, NaiveSSE 22.9

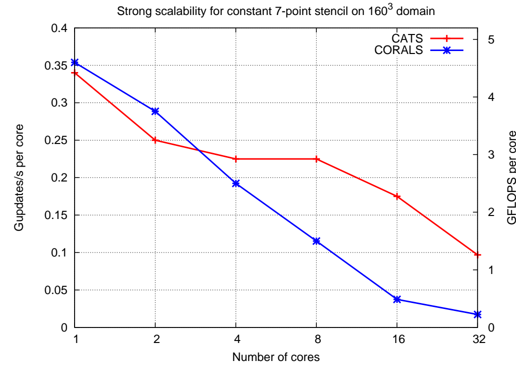



**Figure 7.27.** Constant stencil strong scalability for 1 to 32 threads on a $160^3$ domain of doubles and 100 timesteps on the Xeon X7550. GFLOPS achieved with 32 cores, nuCORALS 104.8, nuCATS 84.5, CATS 40.3, CORALS 7.2, Pochoir 16.9, PLuTo 13, NaiveSSE 44.7

nuCATS and nuCORALS maintain a clear advantage of around 2x over the naive scheme.

The overall performance of nuCATS and nuCORALS grows favorably when more sockets are engaged, while NUMA ignorance can even lead to a drop in the overall performance: for Pochoir from 16 to 32 cores on all domains, for CORALS from 8 to 16 to 32 cores on the $160^3$ domain and from 16 to 32 cores on the $500^3$ domain. The drop in the overall performance of the NaiveSSE scheme occurs already for 8 cores, because the partitioned caches offer less opportunity for data reuse. But since it observes the data-to-core affinity requirement, it scales linearly beyond one NUMA node.

In summary, we see that data-to-core affinity is critical for maintaining performance beyond one NUMA node and also helps on a single socket with a multi-channel bus. Ignorance of the NUMA aspect in today's memory systems can even lead to the situation that a naive scheme which observes this aspect outperforms more sophisticated schemes that ignore it.

## 7.4 Conclusions

Spatio-temporal locality, parallelism, regular memory access and data-to-core affinity are all key requirements to achieve high performance on iterative stencil computations. We have shown that a systematic treatment of these requirements brings forth schemes that deliver high absolute performance and overall good scalability on many-core systems.

Analysis of our previous schemes CATS and CORALS, and other temporal blocking algorithms that do not take data-to-core affinity into account demonstrates a huge per-core slowdown when scaling beyond one NUMA node; sometimes this even results in a drop

of overall performance. Our new schemes nuCORALS and nuCATS on the other hand continue to benefit from additional cores even in the case of strong scaling on a small domain.

# Part III

# Application

# Chapter 8

# Optical Flow Estimation from RGBZ Cameras

Stencil Computations are at the core of many scientific and engineering applications. In this Chapter, we introduce a potential application from the computer vision field, in particular, the scene flow which is mainly a correspondence problem. We use the variational framework to cast the problem as an optimization problem of an energy functional. The minimizer is the solution for the discretized Euler-Lagrange equations which often boils down to an equation system which can be solved numerically by an iterative solver, e.g. Gauss Seidel. The iterative numerical solvers are good examples for stencil computations which can use CATS (Chapter 4), CORALS (Chapter 6), nuCATS and nuCORALS (Chapter 7) to optimize their execution times. However, the adaptation of these schemes for our scene flow algorithm is left as a future work.

Scene flow is the 3D motion field of objects in the scene, as opposed to the optical flow which is the projection of the 3D motion field onto the image plane. Scene flow is of high importance for many computer vision tasks such as vehicle navigation [66] and motion capture [64]. Most existing approaches for computing scene flow either use 2D image data [65] or known 3D scene information [23] as input. Image based scene flow estimation approaches often solve for the depth and the 2D optical flow simultaneously using a stereo setup [29, 66, 64]. In contrast to such methods as well as traditional optical flow approaches that only require colour information, direct 3D scene flow estimation requires knowledge about the depth beforehand [23, 37]. Our approach lies at the boundary of these two categories: It requires the depth information to be available beforehand and uses it in an image based approach to solve for the optical flow from which the scene flow can be easily derived. We assume that a depth map of the scene is available from a time-of-flight camera which is temporally synchronised and spatially calibrated with respect to a colour

camera.

In this Chapter, we propose novel constancy assumptions that can be imposed on the depth map and use them in a global energy functional to solve for the optical flow. To this end, we extend a highly accurate variational optical flow method by incorporating information from the depth sensor. This additional information will lead to a more accurate optical flow than by using colour information alone and will at the same time render the estimation process more stable in regions where depth discontinuities and motion discontinuities coincide. Comparisons to a variational optical flow method that does not use depth information shows the favourable performance of our method at object boundaries and motion discontinuities and thus highlights the advantage of combining depth and colour information in a variational framework. The 3D motion of objects in the scene, i.e. the scene flow, can easily be inferred since the corresponding 3D point of each pixel is known through its depth value.

In summary, our contributions are: (i) The introduction of a framework for combining depth and colour cues in a global variational approach for computing optical flow. (ii) We couple this to a study of suitable invariants derived from the depth map and their combination with colour constancy assumptions. These two contributions will be the topic of Section 8.1 of this paper, where we will also discuss minimisation and implementation details. An experimental evaluation of our ideas on real-world and synthetic data will be presented in Section 8.5.

## 8.1 Optical Flow from RGBZ Images

Our objective is to retrieve the motion of 3D objects in the scene by estimating the optical flow between two consecutive frames of an RGBZ camera, i.e. a combination of a colour and a depth camera. This is possible, since knowing the 2D motion field between two RGBZ images is equivalent to knowing the 3D motion field: The latter can be easily inferred as the change of 3D coordinates that are known in all pixels through their depth value.

### 8.1.1 Setup and Notation

The setup for our method consists of an RGB colour camera and a depth camera, which are assumed to be synchronised and calibrated with respect to each other. We denote by $f_1$, $f_2$ and $f_3$ the red, green and blue output channels of the colour camera and by $f_4$ the output of the depth camera. Because the relative pose and orientation of both cameras is known, the colour and the depth image can be registered into each other. This means

that all channels $f_i$, for $i \in \{1, 2, 3, 4\}$, can be expressed on a common image domain $\Omega \subset \mathbb{R}^2$ and that $(f_1, f_2, f_3, f_4)$ can be regarded as the output of a single RGBZ camera. Since we are dealing with dynamic scenes, each channel is regarded as a scalar-valued image sequence $f_i(x, y, t) : \Omega \times [0, \infty) \to \mathbb{R}$, with $\vec{x} = (x, y)^\top$ a position in $\Omega$ and $t$ the time variable. The optical flow between two consecutive RGBZ images will be denoted by $\vec{w} = (u, v, 1)^\top$. To remove high frequency noise in the input data and to guarantee well-posedness of the optical flow method, $f_i$ is convolved with a small Gaussian kernel of $\sigma \approx 1$.

### 8.1.2 A Variational Model for Optical Flow from RGBZ Images

To estimate the optical flow $\vec{w}$ between two consecutive RGBZ images, we minimise the energy

$$E(\vec{w}) = \int_\Omega \big(E_\mathrm{D}(\vec{w}) + \alpha E_\mathrm{S}(\vec{w})\big)\,\mathrm{d}\vec{x}\ , \tag{8.1}$$

where $E_\mathrm{D}$ is the *data term* that imposes constancy on certain image features and $E_\mathrm{S}$ is the *smoothness term* that imposes regularity on the motion field. The *smoothness weight* $\alpha > 0$ serves as a regularization parameter that controls the relative influence of both terms. In the following we will detail on the design of both terms.

### 8.1.3 The Data Term

To establish correspondences between successive frames, we make use of two successful concepts from variational optical flow literature. First of all, we assume that the intensity value of a scene point does not change over time. As a result, the brightness of the corresponding image points stays constant along the projected motion path. This gives rise to the classical *brightness constancy assumption* [28]. For real-world image sequences, this assumption often does not hold, especially in case of illumination changes. To account for additive illumination changes in the scene, we additionally assume that the brightness gradient does not change along the optical flow trajectory [8]. For a single image channel $f$, both constancy assumptions combined would lead to a data term of the form

$$E_\mathrm{D}(\vec{w}) = \Psi\Big(\,|f(\vec{x}+\vec{w}) - f(\vec{x})|^2 + \gamma\left|\vec{\nabla}_2 f(\vec{x}+\vec{w}) - \vec{\nabla}_2 f(\vec{x})\right|^2\Big)\ , \tag{8.2}$$

where $\gamma$ is a positive weight that steers the influence of the gradient constancy assumption and $\vec{\nabla}_2 = (\partial_x, \partial_y)^\top$ stands for the spatial gradient operator. The function $\Psi(s^2)$ is a convex sub-quadratic penalizer that provides robustness against outliers arising from e.g.

noise and occlusions. We choose it to be the regularized $L_1$-norm

$$\Psi(s^2) = \sqrt{s^2 + \epsilon^2} \quad \text{with} \quad \epsilon > 0 \ . \tag{8.3}$$

**Integrating Colour and Depth Information** In our application, we do not have one, but four image channels. To integrate the colour information into our method, we consider a multi-channel variant of the above data term. To this end, the three colour channels are coupled by summing up their singular contributions to both constancy assumptions.

The depth information, however, can not be integrated in such a straightforward way:

(i) Imposing constancy on the depth channel will most certainly fail because the depth of a scene point can change over time due to object or camera motion. Although not totally invariant under motion, edges in the depth channel, on the other hand, provide a better cue such that assuming constancy of the depth gradient makes more sense as a matching term. We make the first contribution of this paper by proposing the following data term for combined depth and colour channels:

$$\begin{aligned} E_{\mathrm{D}A}(\vec{w}) = \Psi\Big( \sum_{i=1}^{3} |f_i(\vec{x}+\vec{w}) - f_i(\vec{x})|^2 + \gamma \sum_{i=1}^{3} \left|\vec{\nabla}_2 f_i(\vec{x}+\vec{w}) - \vec{\nabla}_2 f_i(\vec{x})\right|^2 \\ + \beta \left|\vec{\nabla}_2 f_4(\vec{x}+\vec{w}) - \vec{\nabla}_2 f_4(\vec{x})\right|^2 \Big) \ . \end{aligned} \tag{8.4}$$

(ii) A second observation is that the depth and the colour channels do not share the same information. Except in those regions where depth and intensity edges coincide, they are generally not correlated. This is why their contributions should be weighted and penalized separately. A separate penalization of constancy assumptions of different image channels has been shown to be advantageous if one assumption produces an outlier [9, 73]. Incorporating this idea leads to an energy of the form

$$\begin{aligned} E_{\mathrm{D}B}(\vec{w}) = \quad & \Psi\Big( \sum_{i=1}^{3} |f_i(\vec{x}+\vec{w}) - f_i(\vec{x})|^2 + \gamma \sum_{i=1}^{3} \left|\vec{\nabla}_2 f_i(\vec{x}+\vec{w}) - \vec{\nabla}_2 f_i(\vec{x})\right|^2 \Big) \\ & + \beta \ \Psi\Big( \left|\vec{\nabla}_2 f_4(\vec{x}+\vec{w}) - \vec{\nabla}_2 f_4(\vec{x})\right|^2 \Big) \ . \end{aligned} \tag{8.5}$$

**An Improved Invariant on the Depth Channel** The depth gradient constancy might be an inappropriate assumption under general 3D motion, since it is bound to a 2D projected image. It is therefore better to use an invariant that is encoded in 3D space, but can still be computed from the 2D depth image. If we assume that the objects in the

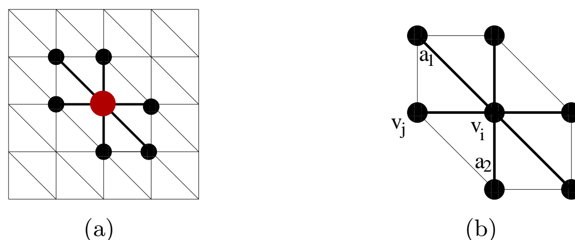



**Figure 8.1.** (a) The 3D mesh topology obtained by triangulation of a depth image. A vertex (red) and its neighbours (black). (b) The laplacian of the vertex $\vec{v_i}$ is computed using the cotangent of the angles $a_1$ and $a_2$ opposite the edge $(v_i, v_j)$.

scene do not change shape too much, the Laplacian coordinates [55] of the associated 3D mesh are a good choice.

The Laplacian coordinates of a vertex $\vec{v_i}$ in a 3D triangular mesh are computed as

$$\vec{\delta^i} = (\delta_i^x, \delta_i^y, \delta_i^z) = \frac{1}{|\Omega_i|} \sum_{j \in N(i)} \frac{1}{2} (\cot \alpha_{ij} + \cot \beta_{ij}) (\vec{v_i} - \vec{v_j}) \ , \tag{8.6}$$

where $N(i)$ is the set of immediate neighbours of $\vec{v_i}$, $|\Omega_i|$ is the size of the Voronoi cell of $\vec{v_i}$ and $\alpha_{ij}$ and $\beta_{ij}$ denote the angles opposite to the edge $(\vec{v_i}, \vec{v_j})$. This is illustrated in Fig. 8.1(b), where $\alpha_{ij}$ and $\beta_{ij}$ correspond to $a_1$ and $a_2$, respectively. In practice, we obtain a 3D mesh of the scene by backprojecting each pixel using its known depth value and the known camera parameters and triangulating the 3D points based on the image grid connectivity as shown in Fig. 8.1(a). As a result, we can compute a Laplacian coordinate $\vec{\delta}(\vec{x})$ in each image point $\vec{x}$.

Since the mesh Laplacian implicitly encodes both local surface orientation and curvature in 3D, it is not rotationally invariant and therefore it only makes sense to assume constancy on its magnitude. We make a second contribution in this paper by proposing a data term that combines colour constancy with Laplacian magnitude constancy as in

$$\begin{aligned} E_{\mathrm{DC}}(\vec{w}) = \Psi\Big( &\sum_{i=1}^{3} |f_i(\vec{x}+\vec{w}) - f_i(\vec{x})|^2 + \gamma_1 \sum_{i=1}^{3} \left|\vec{\nabla}_2 f_i(\vec{x}+\vec{w}) - \vec{\nabla}_2 f_i(\vec{x})\right|^2 \\ &+ |F(\vec{x}+\vec{w}) - F(\vec{x})|^2 + \gamma_2 \left|\vec{\nabla}_2 F(\vec{x}+\vec{w}) - \vec{\nabla}_2 F(\vec{x})\right|^2 \Big) \ , \end{aligned} \tag{8.7}$$

where the Laplacian magnitude channel is defined as $F(\vec{x}) := \|\vec{\delta}(\vec{x})\|$. Note that as opposed to the depth channel $f_4$, both constancy of the value and the gradient are imposed on the Laplacian magnitude channel $F$ for increased robustness. As for the data term (8.5), we can also apply a separate penalization of both colour and Laplacian magnitude assump-

tions:

$$E_{\mathrm{D}D}(\vec{w}) = \Psi\Big( |f(\vec{x}+\vec{w}) - f(\vec{x})|^2 + \gamma_1 \left|\vec{\nabla}_2 f(\vec{x}+\vec{w}) - \vec{\nabla}_2 f(\vec{x})\right|^2 \Big) + \beta\, \Psi\Big( |F(\vec{x}+\vec{w}) - F(\vec{x})|^2 + \gamma_2 \left|\vec{\nabla}_2 F(\vec{x}+\vec{w}) - \vec{\nabla}_2 F(\vec{x})\right|^2 \Big) \ . \tag{8.8}$$

The methods using the above data terms will further be denoted as methods `A`, `B`, `C` and `D`.

### 8.1.4 Smoothness Term.

To penalize deviations from piece-wise smoothness and to preserve semantically important edges in the flow field, we choose the following regularizer

$$\mathcal{E}_{\mathrm{S}}(\vec{w}) = \Psi\left(|\vec{\nabla}_2 \vec{w}|^2\right) = \Psi\left(|\vec{\nabla}_2 u|^2 + |\vec{\nabla}_2 v|^2\right) \ , \tag{8.9}$$

where $\Psi$ is the same sub-quadratic penalizer as in Eq. (8.3). The smoothness term fills in information in regions where the data term does not provide a unique solution and is therefore important in obtaining dense scene flow results.

## 8.2 Minimisation

A minimiser $\vec{w}$ of the energy (8.1) has to be a solution of the associated Euler-Lagrange equations with homogeneous Neumann boundary conditions. At first, we assume that the displacements are small and that both the brightness and the gradient constancy assumptions can be approximated sufficiently well by their first order Taylor expansions. Making use of the *motion tensor notation* [10, 73], the squared linearised brightness difference in (8.4) can be written in a more compact form as

$$|f(\vec{x}+\vec{w}) - f(\vec{x})|^2 \approx |f_x u + f_y v + f_t|^2 = \left|\vec{\nabla}_3^\top f \vec{w}\right|^2 = \vec{w}^\top J \vec{w} \ , \tag{8.10}$$

with $\vec{\nabla}_3 = (\partial_x, \partial_y, \partial_t)^\top$ and $J := \vec{\nabla}_3 f \, \vec{\nabla}_3^\top f$ a $3 \times 3$ tensor. In the same way, the two linearised gradient component differences in (8.2) give rise to the two $3 \times 3$ tensors $J_x := \vec{\nabla}_3 f_x \, \vec{\nabla}_3^\top f_x$ and $J_y := \vec{\nabla}_3 f_y \, \vec{\nabla}_3^\top f_y$. If we now denote by $J_i$, $J_{x,i}$ and $J_{y,i}$ the respective tensors for a specific image channel $f_i$, we can write the data term (8.4) as

$$E_{\mathrm{D}A}(\vec{w}) = \Psi\Big(\vec{w}^\top J_m \vec{w}\Big) \ , \tag{8.11}$$

with the $3 \times 3$ *motion tensor* $J_{mA}$

$$J_{mA} = \sum_{i=1}^{3} J_i + \gamma \sum_{i=1}^{3} (J_{x,i} + J_{y,i}) + \beta (J_{x,4} + J_{y,4}) \quad , \tag{8.12}$$

which combines all constancy assumptions and channel information in the data term (8.4). Similarly, one can obtain a compact form for the other data terms. Due to space limitation; however, we only show the compact form for the data term (8.8)

$$E_{\mathrm{D}D}(\vec{w}) = \Psi\Big(\vec{w}^\top J_m \vec{w}\Big) + \Psi\Big(\vec{w}^\top J_l \vec{w}\Big) \quad , \tag{8.13}$$

with the $3 \times 3$ *motion tensors* $J_{mD}$ *and* $J_{lD}$

$$J_{mD} = \sum_{i=1}^{3} J_{mi} + \gamma_1 \sum_{i=1}^{3} \Big(J_{mx,i} + J_{my,i}\Big) \quad , \tag{8.14}$$

$$J_{lD} = \sum_{i=1}^{3} J_{li} + \gamma_2 \sum_{i=1}^{3} \Big(J_{lx,i} + J_{ly,i}\Big) \quad , \tag{8.15}$$

Using this compact notation, the final Euler-Lagrange equations for the $u$ and the $v$-component of the optical flow of the data term (8.4) can be written as

$$0 = \Psi'\left(\vec{w}^\top J_m \vec{w}\right) (J_{m11} u + J_{m12} v + J_{m13}) - \alpha \, \mathrm{div}\Big(\Psi'\big(|\vec{\nabla}_2 \vec{w}|^2\big) \nabla u\Big), \tag{8.16}$$

$$0 = \Psi'\left(\vec{w}^\top J_m \vec{w}\right) (J_{m12} u + J_{m22} v + J_{m23}) - \alpha \, \mathrm{div}\Big(\Psi'\big(|\vec{\nabla}_2 \vec{w}|^2\big) \nabla v\Big) \quad , \tag{8.17}$$

where $J_{mij}$ stands for the $i,j$-th entry of the motion tensor $J_{mA}$ for the data term (8.4). When the data term (8.8) is used, $J_{mD}$ is used instead of $J_{mA}$ and the following term is added to Equations (8.16) and (8.17)

$$\Psi'\left(\vec{w}^\top J_l \vec{w}\right) (J_{l11} u + J_{l12} v + J_{l13}) \quad , \tag{8.18}$$

## 8.3 Implementation

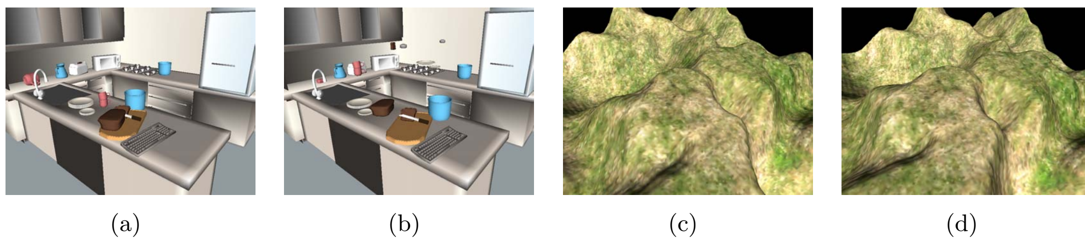
The solution of the Euler-Lagrange equations boils down to solving a nonlinear equation system. To remove the nonlinearity due to the sub-quadratic penalization, a fixed point iteration is applied in which the arguments of $\Psi'(s^2)$ are fixed using the current estimate of the optical flow. The resulting linear system of equations is then solved by successive over-relaxation (SOR) solver which can use our schemes to speed up its computation; however, we leave this as a future work as mentioned before.

To account for large displacements, we apply our solution in a coarse-to-fine multiscale warping framework [8]. To obtain a coarse representation of the problem, we downsample the input images by a factor $\eta \in [0.5, 1.0)$. At each warping level, we split the flow field into an already computed solution from coarser levels and an unknown flow increment. As the increments are small, they can be computed by the presented linearised approach. At the next finer level, the already computed solution serves as initialisation, which is achieved by performing a motion compensation of the second frame by the current flow, known as warping.

## 8.4 Scene Flow Derivation

Until now we have only estimated the optical flow between two consecutive RGBZ images. To derive the scene flow, we add the optical flow vector of each pixel to the corresponding coordinate in the first RGBZ image and perform a bilinear interpolation between the four points in the second image. Scene flow is then computed as the difference between the interpolated 3D coordinate in the second depth map and the 3D coordinates in the first depth map. We have experienced, however, that this procedure can result in erroneous flow vectors at object boundaries due to the interpolation of background and foreground motion. To deal with this problem, we apply a vector median filter in the boundary region as a post-processing step to remove the noisy flow vectors that result from the interpolation.

## 8.5 Evaluation



(a) (b) (c) (d)

**Figure 8.2.** Kitchen sequence: frame (a) 1 and (b) 16. Terrain sequence: frame (c) 1 and (d) 2.

In a first set of experiments, we evaluate our algorithm on two synthetic sequences that we have rendered ourselves in OpenGL. The first sequence shows a kitchen scene where objects move around freely, while the second sequence simulates a camera moving over a rough terrain. Two frames of each sequence are shown in Fig. 8.2. Both sequences have a resolution of $800 \times 600$. The Kitchen sequence is 27 frames long and features small,

localised displacements, while the Terrain sequence contains 7 frames and exhibits large overall rotating motion. For each frame, we generated the ground truth depth map, as well as the ground truth optical flow and scene flow with respect to the next frame. Both synthetic sequences will be made public for research purposes.

In Tab. 8.1 we show a comparison between the baseline method of Brox et al. [8] and the variants of our method with the different data terms proposed in Section 8.1.3. We report the *average angular error (AAE)* for both the 2D optical flow [3] and the 3D scene flow. The 3D average angular error for scene flow can be defined as an extension of the 2D AAE [66]

$$AAE_{3D} = \arccos \frac{(u_{c1}u_{e1} + u_{c2}u_{e2} + u_{c3}u_{e3} + 1)}{\sqrt{\left(u_{c1}^2 + u_{c2}^2 + u_{c3}^2\right)\left(u_{e1}^2 + u_{e2}^2 + u_{e3}^2\right)}} \quad , \tag{8.19}$$

where $(u_{c1}, u_{c2}, u_{c3})$ and $(u_{e1}, u_{e2}, u_{e3})$ denote ground truth and the estimate scene flow. The values reported in the table are the average AAE taken over all frames of the respective sequence and the standard deviation. From the table it is clear that the idea of incorporating depth information in optical flow leads to an improvement of both the estimated 2D and 3D motion. Moreover, the methods `C` and `D`, which assume constancy on the mesh Laplacian, achieve the best results on both sequences. This illustrates that a careful design of motion invariants in the 3D domain can lead to an improvement of the optical flow.

We also apply our algorithm on two real-world sequences that have been provided by the authors of [50]. These sequences have been recoded by a Point Grey Flea2 camera and a Swiss Ranger 4000 time-of-flight camera, which have been calibrated with respect to each other. For our purpose, the depth images are upsampled from a resolution of $176 \times 144$ to the resolution of the colour images using the method described in [50].

An example of two frames of the first sequence is shown in Fig. 8.3. Here, a hand is moving up against a static untextured background. The figure also shows the motion estimated with the method of Brox et al. and the best result that we obtained with our method, which corresponds to method `D`. While the base-line method clearly oversmooths depth discontinuities, the combined depth and colour method achieves visually sharper results. Unlike for the synthetic sequences, best results were obtained with method `D`, which incorporates Laplacian magnitude constancy and seperate penalization. The second row of Fig. 8.3 shows 3D scene flow results corresponding to the two optical flow results. Here we can see that our method achieves a better estimate of the motion in the background and a sharper motion boundary.

For the second sequence in Figure 8.4, the improvement using the method `D` is demonstrated best in the top left corner of the flow field. Method `D` shows no flow here because

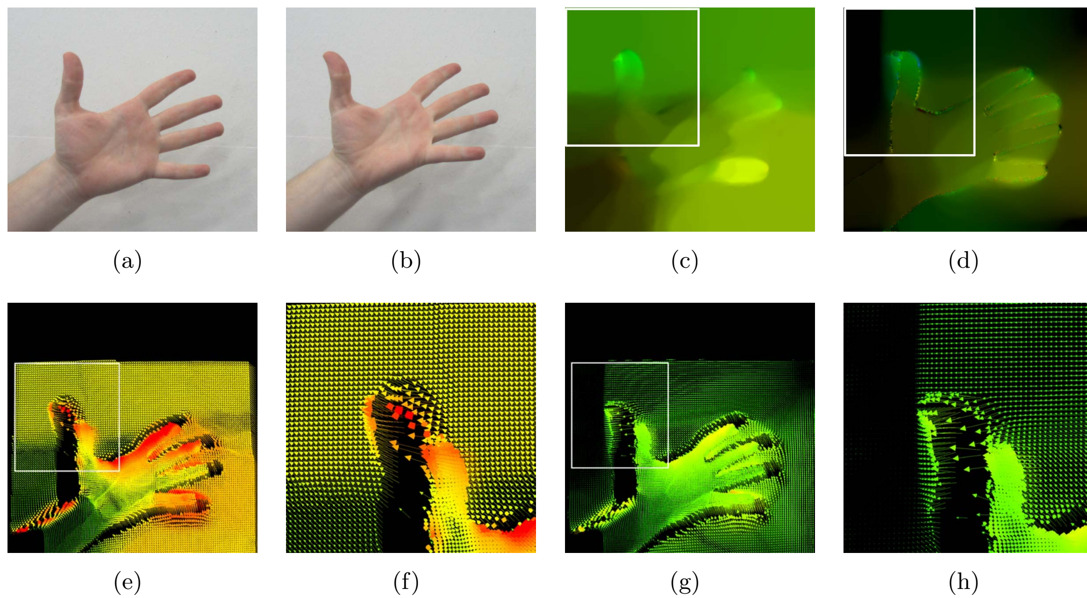



**Figure 8.3.** Hand sequence: frame (a) 1 and (a) 2. Optical flow results using (c) Brox *et al.* [8] and (d) combined depth and colour method `D`. Color intensity encodes absolute flow magnitude. Scene flow results using (e) Brox *et al.* and (g) method `D`. Color intensity encodes relative flow magnitude. Any color difference between the two results is due to a difference in the estimated scene flow range. Zoom in (f) and (h).

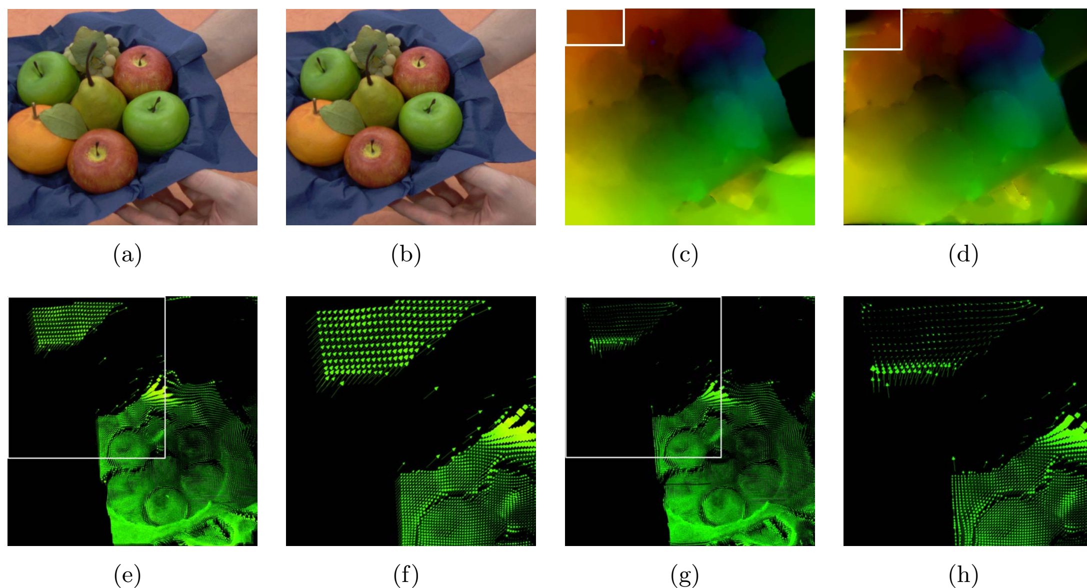



**Figure 8.4.** Fruit sequence: frame (a) 1 and (b) 2. Optical flow results using (c) Brox *et al.* and (d) combined depth and colour method `D`. Scene flow results using (e) Brox *et al.* and (g) method `D`. Zoom in (f) and (h). Note how the background is again estimated better.

it is part of the static background, whereas the base-line method oversmooths. Figure 8.4 also shows the 3D scene flow. As in the case of the previous sequence and the synthetic

**Table 8.1.** A comparison of different scene flow methods on synthetic Terrain sequence (left table) and Kitchen sequence (right table). AAE stands for average angular error. 2D denotes the optical flow error and 3D denotes the scene flow error. Methods A, B, C, and D represent our method with the different data terms (8.4), (8.5), (8.7), and (8.8), respectively.

| Method | $AAE_{2D}$ | $\sigma_{2D}$ | $AAE_{3D}$ | $\sigma_{3D}$ |
|---|---|---|---|---|
| Brox *et al.* | 3.87 | 0.68 | 1.20 | 0.09 |
| Method A | 3.84 | 0.55 | 1.19 | 0.11 |
| Method B | 3.74 | 0.42 | 1.08 | 0.09 |
| Method C | **3.24** | **0.23** | **0.76** | **0.12** |
| Method D | 3.33 | 0.32 | 0.78 | 0.15 |

| Method | $AAE_{2D}$ | $\sigma_{2D}$ | $AAE_{3D}$ | $\sigma_{3D}$ |
|---|---|---|---|---|
| Brox *et al.* | 4.42 | 1.3 | 0.37 | 0.13 |
| Method A | 4.44 | 1.33 | 0.37 | 0.12 |
| Method B | 4.41 | 1.6 | 0.36 | 0.11 |
| Method C | **3.32** | **0.89** | **0.34** | **0.12** |
| Method D | 3.35 | 0.80 | 0.35 | **0.12** |

sequences, these results show that including the mesh Laplacian gives the best overall results.

## 8.6 Conclusion

We have presented a variational framework for optical flow estimation which combines depth information with the traditional colour brightness and gradient constancy assumption. To this end we have proposed four different choices for imposing constancy on depth data: First we proposed a gradient constancy constraint on the depth channel, which could be either penalized jointly with the colour constancy or separately. Secondly, we proposed to impose constancy on the magnitude of the mesh Laplacian, which is invariant under 3D rigid motion. Experimental results on synthetic and real-world RGBZ image sequences have confirmed that a careful design of motion invariants in the 3D domain leads to better optical flow estimates. We have also shown that including depth information can generally avoid oversmoothing at depth continuities.

# Chapter 9

# Conclusion and Future Work

The exponential growth of cores on CPUs leads to exacerbating the memory wall problem where limited off-chip bandwidth capabilities severely restrict the performance of stencil computations. The main motivation of the work presented in this thesis is to lay out the requirements to achieve high performance stencil computations on modern architectures and to design stencil algorithms that comply with these requirements. In particular, we overcome this problem by computational schemes that scale mainly with the aggregate cache bandwidth rather than the system bandwidth. While at first this seems impossible for gigabyte large domains that can never fit into caches, our cache accurate time skewing schemes (CATS) and cache oblivious parallelograms (CORALS) presented in Chapter 4 and Chapter 6, respectively, do deliver this type of strong scalability for certain stencil computations. The challenge here is that the requirements are often conflicting which renders it cumbersome to devise such algorithms that reckon with all requirements simultaneously.

Clearly, an algorithm operating repeatedly on gigabyte large domains cannot become totally independent of the system bandwidth as the data must be read multiple times from system memory. But for certain iterative stencil computations, CATS and CORALS scale very favorably with the increased cache bandwidth and is applicable to both constant and variable stencil problems, i.e. iterative applications of sparse banded matrices are supported. Furthermore, CATS can be applied to stencils of any size and order as long as the problem in hand is bandwidth bound. On the other hand, CORALS demonstrates extraordinary performance especially on the Xeon architecture, it particularly approaches the performance of a synthetic on-chip benchmark for certain stencil computations in 2D, and thus virtually breaks the dependence on the slow off-chip connection. This is a highly desired feature, specifically, for the many-core architectures that exhibit an even larger discrepancy between the on-chip and off-chip bandwidth due to the exponential growth

of CPU cores. On 3D domains, the performance of CORALS is less astounding but still clearly superior to the performance of the general parallelizer and locality optimizer PluTo and the heavily optimized naive scheme. This is an expected result from a more specialized cache oblivious algorithm, but has not been demonstrated before. Stencil computations may involve additional data from other vectors, so typical iterative linear equation solvers like Jacobi or Gauss-Seidel solver can be accelerated using CATS and CORALS.

In order to study the sensitivity of iterative stencil computations to system and cache bandwidths, we develop a performance model for both the CATS and the naive schemes in Chapter 5. The schemes exhibit almost entirely opposite behavior. While the naive scheme demands high system bandwidth for performance, the same stencil computation can be performed with a time skewing scheme much faster if only the cache bandwidth in the CPU is increased. The latter option gives by far the more cost-efficient performance gains, e.g. we could execute on the ten years old Xeon MP as fast as on a Core i7 940 if only sufficient cache bandwidth in the Xeon MP were provided without the need for any improvement of its outdated system bus. The paradoxical conclusion is that for iterative stencil computations further deteriorating the ratio of off-chip to on-chip bandwidth is the cheapest way to higher performance. Unfortunately, the situation is more complex in practice because not all stencil computations occur in iterations and many of them operate with varying rather than constant coefficients which puts additional strain on the system bus. Although the performance model is restricted to iterative stencil computations, it is clear that a solution to the bandwidth wall problem should not be sought solely in scaling the system bandwidth, because it is not necessarily the limiting factor even if the data is much bigger than the caches and has to be accessed many times.

Many-core architectures are quickly becoming the mainstream in computing. These architectures are often equipped with non uniform memory access (NUMA) to improve the memory access scalability. Although this new technology has been introduced to improve the memory access scalability in the older symmetric multiprocessing (SMP) systems, ignoring the fact that memory access time depends on the memory location relative to the processor can deteriorate the absolute performance and the performance scalability of parallel programs. In other words, the latter technology puts a new challenge on applications to achieve high performance on the NUMA architectures. This challenge can be faced by ensuring the data-to-core affinity. In Chapter 7, we show that CATS and CORALS hardly scale beyond one socket on two many-core architectures. Further, we show how the algorithmic building blocks of CORALS and CATS can be systematically adapted to cope with this essential evolution and to reckon with the data-to-core affinity performance requirements. Results show that nuCATS and nuCORALS, in contrast to CATS and CORALS, demonstrate weak as well as strong scalability on two 16- and 32-core NUMA machines.

Comparing the performance of the cache-aware nuCATS against the cache-oblivious nuCORALS in Chapter 7, we have observed that the latter scheme performs better on smaller domains, for which a higher fraction of data reuse occurs in higher level caches, and the former is better on large domains, on which the reduction of the problem-dimension by the wavefront traversal creates far fewer cache misses. Moreover, nuCORALS automatically exploits deep cache hierarchies like those on the Xeon, while nuCATS relies mainly on the performance of the last cache level and therefore nuCATS wins the comparison on a machine which has a shallow cache hierarchy.

In summary, spatio-temporal data locality, parallelism, regular memory access and data-to-core affinity are all key requirements to achieve high performance on iterative stencil computations. We show that a systematic treatment of these requirements brings forth schemes that deliver high absolute performance and overall excellent scalability on many-core systems. Analysis of our previous schemes CATS and CORALS, and other temporal blocking algorithms that do not take data-to-core affinity into account demonstrates a huge per-core slowdown when scaling beyond one NUMA node; sometimes this even results in a drop of overall performance. Our new schemes nuCORALS and nuCATS on the other hand continue to benefit from additional cores even in the case of strong scaling on a small domain.

Diverse areas can benefit from our schemes such as the iterative algorithms in image denoising, segmentation, and registration, optical flow estimation, and physical simulations. As a candidate application, we present a novel variational framework for optical flow estimation which combines depth information with the traditional color brightness and gradient constancy assumption in Chapter 8. We have shown four different choices for imposing constancy on depth data: First we have proposed a gradient constancy constraint on the depth channel, which could be either jointly penalised with the colour constancy or separately. Secondly, we have proposed to impose constancy on the magnitude of the mesh Laplacian, which is invariant under 3D rigid motion. Experimental results on synthetic and real-world RGBZ image sequences demonstrate that a careful design of motion invariants in the 3D domain leads to better optical flow estimates. We have also shown that including depth information can generally avoid oversmoothing at depth continuities.

Our work advances the state of the art in several directions. In cache aware stencil computations, CATS proposes a novel usage of a wavefront traversal in multi-dimensional time skewing, an unconventional departure from the complexity of the commonly used techniques of multi-dimensional tiling and multi-level tiling. Apart from its simplicity, this novel strategy is particularly successful on stencils of order one, where the algorithm breaks the dependence on the low system bandwidth and achieves at least 50% of the stencil peak benchmark performance in 2D and 3D even when operating on gigabyte large domains. This is a significant improvement over a heavily optimized naive scheme and

the state-of-art in automatic optimization. For large stencils and banded matrices the system bandwidth limits the performance again but in comparison CATS maintains a clear advantage.

In cache oblivious stencil computations, CORALS shows how to translate the enormous reduction of cache misses which characterizes cache oblivious schemes into performance. In particular, CORALS uses a tiling structure that caters for the regularity of memory accesses alongside the data locality, and parallelism. Using parallelograms only as the base locality elements in CORALS yields this regularity in memory accesses. The last property helps to avoid adding control logic at the base element level which could exceed the overhead of the actual computation itself. On the other hand, this enables the compiler to use SIMD instructions for the computation. Despite its importance, the regularity of memory accesses has been overlooked in the previous cache oblivious stencil computation and thus makes CORALS the first to achieve the realization of performance from the huge reduction of the cache miss rate.

Further on, we lay out the essential aspects that must be considered in any stencil implementation to achieve high performance on ccNUMA architectures. While previous approaches consider only a subset of these measures, nuCATS and nuCORALS show how to devise successful schemes that cater for all these conflicting measures simultaneously. The high absolute performance and its scalability beyond one NUMA node on many core architectures were not possible with previous approaches.

Our variational optical flow framework enables a high accurate optical flow estimates which are smooth, yet sharp at depth discontinuities. The latter is a highly desirable feature for many image processing and computer vision tasks. This is relatively tedious with previous approaches because in order to obtain sharp estimates at depth discontinuities, it is required to reduce the weight on the smoothness term which could result in lowering the overall smoothness of the optical flow estimates not only at depth discontinuities, but also inside object boundaries.

Future work for any stencil computation scheme will always be to make it faster and applicable to a larger class of stencil computation problems, e.g. those involving unstructured grids. However, there are various more specific directions that could be researched.

The domain decomposition in the nuCATS and nuCORALS schemes can result in unbalanced load between the subdomains when the schemes are run with non-power of 2 number of threads. A future work would be to modify it to take into account balancing the work in all subdomains. This work-subdomain balance can be realized as a preprocessing step using a load balancing structure similar to the one used in CORALS. This not only enables running nuCATS and nuCORALS with any number of threads, but also makes the schemes applicable to a wider class of domains, e.g. circular domains. On the other hand,

nuCATS and nuCORALS can easily be adapted for distributed systems wherein there is no single shared address space. In this case, areas to which thread access has to be synchronized in nuCORALS have to be explicitly transfered over the network between the computing nodes in the cluster. The applicability of nuCATS and nuCORALS can be expanded if they are enclosed in a compiler or automatic transformation tool such as Pochoir [63] and PluTo [7].

# Bibliography


[1] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *In Proc. of SC 2000*, page 31, 2000.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. D. Vorst. Templates for the solution of linear systems: Building blocks for iterative methods, 1994.

[3] J. L. Barron, D. J. Fleet, and S. S. Beauchemin. Performance of optical flow techniques. 12(1):43–77, Feb. 1994.

[4] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parametrized tiling revisited. In *Proc. of the International Symposium on Code Generation and Optimization (CGO'10)*, 2010.

[5] L. N. Bhuyan, H. Wang, and R. Iyer. Impact of cc-numa memory management policies on the application performance of multistage switching networks. *IEEE Trans. Parallel Distrib. Syst.*, 11(3):230–246, Mar. 2000.

[6] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. Technical report, Carnegie Mellon University, 2009.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008.

[8] T. Brox, A. Bruhn, N. Papenberg, and J. Weickert. High accuracy optic flow estimation based on a theory for warping. In T. Pajdla and J. Matas, editors, *Computer Vision – ECCV 2004*, volume 3024 of *Lecture Notes in Computer Science*, pages 25–36. Springer, Berlin, 2004.

[9] A. Bruhn and J. Weickert. Towards ultimate motion estimation: Combining highest accuracy with real-time performance. In *Proc. Tenth International Conference on Computer Vision*, volume 1, pages 749–755, Beijing, China, June 2005. IEEE Computer Society Press.

[10] A. Bruhn, J. Weickert, T. Kohlberger, and C. Schnörr. A multigrid platform for real-time motion computation with discontinuity-preserving variational methods. *International Journal of Computer Vision*, 70(3):257–277, Dec. 2006.

[11] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society.

[12] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart. Parallel data-locality aware stencil computations on modern micro-architectures. In *IPDPS*, pages 1–10, 2009.

[13] K. Datta. Auto-tuning stencil codes for cache-based multicore platforms, 2009.

[14] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159.

[15] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Press, 2008.

[16] H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 642–653, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[18] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 271–280, New York, NY, USA, 2006. ACM.

[19] M. Frigo and V. Strumpen. The memory behavior of cache oblivious stencil computations. *J. Supercomput.*, 39(2):93–112, 2007.

[20] M. A. Frumkin and R. F. Van der Wijngaart. Tight bounds on cache use for stencil operations on rectangular grids. *Journal of ACM*, 49(3):434–453, 2002.

[21] R. P. Garg and I. Sharapov. *Techniques for Optimizing Applications: High Performance Computing*. Prentice Hall Professional Technical Reference, 2002.

[22] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. University of Passau, June 2004. Habilitation thesis.

[23] S. Hadfield and R. Bowden. Kinecting the dots: Particle based scene flow from depth sensors. In D. N. Metaxas, L. Quan, A. Sanfeliu, and L. Van Gool, editors, *Proc. Thirteenth International Conference on Computer Vision*, pages 2290–2295, Barcelona, Nov. 2011. IEEE Computer Society Press.

[24] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. Loop transformation recipes for code generation and auto-tuning.

[25] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 147–157, 2009.

[26] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[27] HiTLoG: Hierarchical tiled loop generator. http://www.cs.colostate.edu/MMAlpha/tiling/.

[28] B. Horn and B. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[29] F. Huguet and F. Devernay. A variational method for scene flow estimation from stereo sequences. In *Proc. Eleventh International Conference on Computer Vision*, Rio de Janeiro, Oct. 2007. IEEE Computer Society Press.

[30] H. Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM, 1981.

[31] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[32] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60. ACM, 2006.

[33] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MEMORY SYSTEM PERFORMANCE*, pages 36–43. ACM, 2005.

[34] D. Kim, L. Renganarayanan, D. Rostron, S. V. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 51, 2007.

[35] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244, 2007.

[36] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[37] A. Letouzey, B. Petit, and E. Boyer. Scene flow from depth and color images. In *Proc. 22nd British Machine Vision Conference*, Dundee, England, Sept. 2011. British Machine Vision Association.

[38] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGE SYSTEMS*, 26:2004, 2004.

[39] L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *Proceedings ACM symposium on Principles and practice of parallel programming*, PPoPP '10, pages 213–222, 2010.

[40] J. Mccalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical report, In http://www.haverford.edu/cmsc/davew/cache-opt/cache-opt.html, 1999.

[41] K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations: An Introduction*. Cambridge University Press, New York, NY, USA, 2005.

[42] F. Mueller. Pthreads library interface, 1994.

[43] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, 2010.

[44] D. Orozco and G. Gao. Mapping the FDTD application to many-core chip architectures. Technical report, University of Delaware, Mar. 2009.

[45] PluTo: A polyhedral automatic parallelizer and locality optimizer for multicores. http://-sourceforge.net/projects/pluto-compiler/.

[46] PrimeTile: A parametric multi-level tiler for imperfect loop nests. http://-primetile.sourceforge.net.

[47] H. Prokop. Cache-oblivious algorithms, 1999.

[48] R. Rao, J. Wenck, D. Franklin, R. Amirtharajah, and V. Akella. Segmented bitline cache: Exploiting non-uniform memory access patterns. In *HiPC*, pages 123–134, 2006.

[49] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye. Towards optimal multi-level tiling for stencil computations. In *Proceedings of International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2007.

[50] C. Richardt, C. Stoll, N. A. Dodgson, H.-P. Seidel, and C. Theobalt. Coherent spatiotemporal filtering, upsampling and rendering of RGBZ videos. In *Proceedings of Eurographics*, 2012. To Appear.

[51] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations, 2000.

[52] M. Shaheen and R. Strzodka. Numa aware iterative stencil computations on many-core systems. In *IPDPS*, pages 461–473, 2012.

[53] Y. Song and Z. Li. A compiler framework for tiling imperfectly-nested loops. In *In Proceedings of the Twelfth International Workshop on Languages and Compilers for Parallel Computing*, pages 185–200. Springer-Verlag, 1999.

[54] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[55] O. Sorkine. Differential representations for mesh processing. *Computer Graphics Forum*, 25(4):789–807, 2006.

[56] V. Strumpen and M. Frigo. Software engineering aspects of cache oblivious stencil computations. Technical report, IBM Research, 2006.

[57] R. Strzodka and M. Shaheen. Impact of system and cache bandwidth on stencil computations across multiple processor generations, 2011.

[58] R. Strzodka, M. Shaheen, and D. Pajak. Time skewing made simple. In *Proceedings ACM symposium on principles and practice of parallel programming*, PPoPP '11, Feb. 2011.

[59] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 49–59. ACM, 2010.

[60] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache accurate time skewing in iterative stencil computations. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Sept. 2011.

[61] A. Taflove and S. C. Hagness. *Computational electrodynamics: the finite-difference time-domain method*. Artech House, Norwood, 3rd edition, 2005.

[62] A. S. Tanenbaum and J. R. Goodman. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 1998.

[63] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *SPAA*, pages 117–128. ACM, 2011.

[64] L. Valgaerts, A. Bruhn, H. Zimmer, J. Weickert, C. Stoll, and C. Theobalt. Joint estimation of motion, structure and geometry from stereo sequences. In K. Daniilidis, P. Maragos, and N. Paragios, editors, *Computer Vision – ECCV 2010*, volume 6314, pages 568–581. Springer, Berlin, 2010.

[65] S. Vedula, S. Baker, P. Rander, R. T. Collins, and T. Kanade. Three-dimensional scene flow. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(3):475–480, Mar. 2005.

[66] A. Wedel, C. Rabe, T. Vaudrey, T. Brox, U. Franke, and D. Cremers. Efficient dense scene flow from sparse or dense stereo data. In D. Forsyth, P. Torr, and A. Zisserman, editors, *Computer Vision – ECCV 2008*, volume 5302, pages 739–751. Springer, Berlin, 2008.

[67] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *Proc. IEEE International Computer Software and Applications Conference (COMPSAC'09)*, 2009.

[68] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the cell processor. *Int. J. Parallel Program.*, 35(3):263–298, June 2007.

[69] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for . . ., 2009.

[70] M. Wittmann, G. Hager, and G. Wellein. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. In *Proc. Workshop on Large-Scale Parallel Processing (LSPP'10) at IPDPS'10*, 2010.

[71] M. Wolf. More iteration space tiling. In *Proceedings of Supercomputing '89*, 1989.

[72] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2000.

[73] H. Zimmer, A. Bruhn, and J. Weickert. Optic flow in harmony. *International Journal of Computer Vision*, 93(3):368–388, Apr. 2011.